FabULous Interoperability for ML and a Linear Language

Gabriel Scherer, Max New, Nick Rioux, Amal Ahmed

FabULous Interoperability for ML and a Linear Language

Gabriel Scherer[1,2], Max New[1], Nick Rioux[1], and Amal Ahmed[1,3]

[1] Northeastern University, US
[2] INRIA Saclay, France
[3] INRIA Paris, France

Abstract. Instead of a monolithic programming language trying to cover all features of interest, some programming systems are designed by combining together simpler languages that cooperate to cover the same feature space. This can improve usability by making each part simpler than the whole, but there is a risk of *abstraction leaks* from one language to another that would break expectations of the users familiar with only one or some of the involved languages.

We propose a formal specification for what it means for a given language in a multi-language system to be usable without leaks: it should embed into the multi-language in a *fully abstract* way, that is, its contextual equivalence should be unchanged in the larger system.

To demonstrate our proposed design principle and formal specification criterion, we design a multi-language programming system that combines an ML-like statically typed functional language and another language with linear types and linear state. Our goal is to cover a good part of the expressiveness of languages that mix functional programming and linear state (ownership), at only a fraction of the complexity. We prove that the embedding of ML into the multi-language system is fully abstract: functional programmers should not fear abstraction leaks. We show examples of combined programs demonstrating in-place memory updates and safe resource handling, and an implementation extending OCaml with our linear language.

1 Introduction

Feature accretion is a common trend among mature but actively evolving programming languages, including C++, Haskell, Java, OCaml, Python, and Scala. Each new feature strives for generality and expressiveness, and may provide a large usability improvement to users of the particular problem domain or programming style it was designed to empower (e.g., XML documents, asynchronous communication, staged evaluation). But feature creep in general-purpose languages may also make it harder for programmers to master the language as a whole, degrade the user experience (e.g., leading to more cryptic error messages), require additional work on the part of tooling providers, and lead to fragility in language implementations.

A natural response to increased language complexity is to define subsets of the language designed for a better programming experience. For instance, a subset can be easier to teach (e.g., "Core" ML[4], Haskell 98 as opposed to GHC Haskell, Scala mastery levels[5]); it can facilitate static analysis or decrease the risk of programming errors, while remaining sufficiently expressive for the target users' needs (e.g., MISRA C, Spark/Ada); it can enforce a common style within a company; or it can be designed to encourage a transition to deprecate some ill-behaved language features (e.g., strict Javascript).

Once a subset has been selected, it may be the case that users write whole programs purely in the subset (possibly using tooling to enforce that property), but programs will commonly rely on other libraries that are not themselves implemented in the same subset of the language. If users stay in the subset while using these libraries, they will only interact with the part of the library whose interface is expressible in the subset. But does the behavior of the library respect the expectations of users who only know the subset? When calling a function from within the subset breaks subset expectations, it is a sign of *leaky abstraction*.

How should we design languages with useful subsets that manage complexity and avoid abstraction leaks?

We propose to look at this question from a different, but equivalent, angle: instead of designing a single big monolithic language with some nicer subsets, we propose to consider *multi-language* programming systems where several smaller programming languages interact together to cover the same feature space. Each language or sub-combination of languages is a subset, in the above sense, of the multi-language, and there is a clear definition of *abstraction leaks* in terms of user experience: a user who only knows some of the languages of the system should be able to use the multi-language system, interacting with code written in the other languages, without have their expectations violated. If we write a program in Java and call a function that, internally, is implemented in Scala, there should be no surprises—our experience should be the same as when calling a pure Java function. Similarly, consider the subset of Haskell that does not contain `IO` (input-output as a type-tracked effect): the expectations of a user of this language, for instance in terms of valid equational reasoning, should not be violated by adding `IO` back to the language—in the absence of the abstraction-leaking `unsafePerformIO`.

We propose a *formal specification* for a "no abstraction leaks" guarantee that can be used as a design criterion to design new multi-language systems, with graceful interoperation properties. It is based on the formal notion of *full abstraction* which has previously been used to study the denotational semantics of programming languages [Meyer and Sieber, 1988, Milner, 1977, Cartwright and Felleisen, 1992, Jeffrey and Rathke, 2005, Abramsky, Jagadeesan, and Malacaria, 2000], and the formal property of compilers [Ahmed and Blume, 2008, 2011, Devriese, Patrignani, and Piessens, 2016, New, Bowman, and Ahmed, 2016, Patrignani, Agten, Strackx, Jacobs, Clarke, and Piessens, 2015], but not for user-facing languages. A compiler C from a source language S to a target language T

[4] https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html

[5] http://www.scala-lang.org/old/node/8610

is *fully abstract* if, whenever two source terms s_1 and s_2 are indistinguishable in S, their translations $C(s_1)$ and $C(s_2)$ are indistinguishable in T. In a multi-language $G+E$ formed of a general-purpose, user-friendly language G and a more advanced language E—one that provides an *e*scape hatch for *e*xperts to write code that can't be implemented in G—we say that E does not *leak* into G if the embedding of G into the multi-language $G+E$ is fully abstract.

To demonstrate that our formal specification is reasonable, we design a novel multi-language programming system that satisfies it. Our multi-language λ^{UL} combines a general-purpose functional programming language λ^{U} (unrestricted) of the ML family with an advanced language λ^{L} (linear) with *linear types* and linear state. It is less convient to program in λ^{L}'s restrictive type system, but users can write programs in λ^{L} that could not be written in λ^{U}: they can use linear types, locally, to enforce resource usage protocols (typestate), and they can use linear state and the linear ownership discipline to write programs that do in-place update to allocate less memory, yet remain observationally pure.

Consider for example the following mixed-language program. The blue fragments are written in the general-purpose, user-friendly functional language, while the red fragments are written in the linear language. The boundaries UL and LU allow switching between languages. The program reads all lines from a file, accumulating them in a list, and concatenating it into a single string when the end-of-file (EOF) is reached.

```
let concat_lines path : String = UL(
  loop (open LU(path)) LU(Nil)
  where rec loop handle LU(acc : List String) =
    match line handle with
    | Next line LU(handle) -> loop handle LU(Cons line acc)
    | EOF handle -> close handle; LU(rev_concat "\n" acc))
```

The linear type system ensures that the file handle is properly closed: removing the close handle call would give a type error. On the other hand, only the parts concerned with the resource-handling logic need to be written in the red linear language; the user can keep all general-purpose logic (here, how to accumulate lines and what to do with them at the end) in the more convenient general-purpose blue language—and call this function from a blue-language program. Fine-grained boundaries allow users to rely on each language's strength and to use the advanced features only when necessary.

In this example, the file-handle API specifies that the call to line, which reads a line, returns the data at type ![String]. The latter represents how U values of type String can be put into a *lump* type to be passed to the linear world where they are treated as opaque blackboxes that must be passed back to the ML world for consumption. For other examples, such as in-place list manipulation or transient operations on an persistent data structure, we will need a deeper form of interoperability where the linear world creates, dissects or manipulates U values. To enable this, our multi-language supports translation of types from one language to the other, using a *type compatibility* relation $\sigma \simeq \sigma$ between λ^{U} types σ and λ^{L} types σ.

We claim the following contributions:

1. We propose a formal specification of what it means for advanced language features to be introduced in a (multi-)language system without introducing a class of abstraction leaks that break equational reasoning. This specification captures a useful *usability* property, and we hope it will help us and others design more usable programming languages, much like the formal notion of *principal types* served to better understand and design type inference systems.
2. We design a simple linear language, λ^{L}, that supports linear state (Section 2). This simple design for linear state is a contribution of its own. A nice property of the language (shared by some other linear languages) is that the code has both an imperative interpretation—with in-place memory update, which provides resource guarantees—and a functional interpretation—which aids program reasoning. The imperative and functional interpretations have different resource usage, but the same input/output behavior.
3. We present a multi-language programming system λ^{UL} combining a core ML language, λ^{U} (U for Unrestricted, as opposed to Linear) with λ^{L} and prove that the embedding of the ML language λ^{U} in λ^{UL} is fully abstract (Section 3). Moreover, the multi-language is designed to ensure that our full abstraction result is stable under extension of the embedded ML language λ^{U}.
4. We define a logical relation and prove parametricity for λ^{UL}. The logical relation illustrates, semantically, why one can reason functionally about programs in λ^{UL} despite the presence of state and strong updates (Section 4).
5. We evaluate the resulting language design by providing examples of hybrid λ^{UL} programs that exhibit programming patterns inaccessible to ML alone, such that safe in-place updates and typestate-like static protocol enforcement (Section 5).

2 The λ^{U} and λ^{L} languages

$$
\begin{array}{llll}
\textit{Types} & \sigma & ::= & \alpha \mid \sigma_1 \times \sigma_2 \mid 1 \mid \sigma_1 \to \sigma_2 \mid \sigma_1 + \sigma_2 \mid \mu\alpha.\,\sigma \mid \forall\alpha.\,\sigma \\
\textit{Expr.} & \mathsf{e} & ::= & \mathsf{x} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \pi_1\,\mathsf{e} \mid \pi_2\,\mathsf{e} \mid \langle\rangle \mid \mathsf{e}_1 ; \mathsf{e}_2 \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \\
 & & & \mathsf{inj_i}\,\mathsf{e} \mid \mathsf{case}\,\mathsf{e}'\,\mathsf{of}\,\mathsf{x}_1.\,\mathsf{e}_1 \,|\, \mathsf{x}_2.\,\mathsf{e}_2 \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{e} \mid \mathsf{unfold}\,\mathsf{e} \mid \Lambda\alpha.\,\mathsf{e} \mid \mathsf{e}\,[\sigma] \\
\textit{Values} & \mathsf{v} & ::= & \mathsf{x} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle \mid \langle\rangle \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} \mid \mathsf{inj_1}\,\mathsf{v} \mid \mathsf{inj_2}\,\mathsf{v} \mid \mathsf{fold}_{\mu\alpha.\sigma}\,\mathsf{v} \mid \Lambda\alpha.\,\mathsf{v} \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x}:\sigma \mid \Gamma, \alpha
\end{array}
$$

Fig. 1. Unrestricted Language: Syntax

The unrestricted language λ^{U} is a run-of-the-mill idealized ML language with functions, pairs, sums, iso-recursive types and polymorphism. It is presented in its explicitly typed form—we will not discuss type inference in this work. The full syntax is described in Figure 1, and the typing rules in Figure 2. The dynamic semantics is completely standard. Having binary sums, binary products and iso-recursive types lets us express algebraic datatypes in the usual way.

$\boxed{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma}$

$$\frac{\mathsf{x}{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{U}} \mathsf{x} : \sigma} \qquad \frac{}{\Gamma \vdash_{\mathsf{U}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : 1 \qquad \Gamma \vdash_{\mathsf{U}} \mathsf{e}' : \sigma}{\Gamma \vdash_{\mathsf{U}} \mathsf{e}; \mathsf{e}' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e}_1 : \sigma_1 \qquad \Gamma \vdash_{\mathsf{U}} \mathsf{e}_2 : \sigma_2}{\Gamma \vdash_{\mathsf{U}} \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\mathsf{U}} \pi_{\mathsf{i}}\, \mathsf{e} : \sigma_i}$$

$$\frac{\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathsf{U}} \mathsf{e} : \sigma'}{\Gamma \vdash_{\mathsf{U}} \lambda(\mathsf{x} : \sigma).\, \mathsf{e} : \sigma \to \sigma'} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma' \to \sigma \qquad \Gamma \vdash_{\mathsf{U}} \mathsf{e}' : \sigma'}{\Gamma \vdash_{\mathsf{U}} \mathsf{e}\ \mathsf{e}' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma_i}{\Gamma \vdash_{\mathsf{U}} \mathsf{inj_i}\, \mathsf{e} : \sigma_1 + \sigma_2} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma_1 + \sigma_2 \qquad \begin{array}{c}\Gamma, \mathsf{x}_1 : \sigma_1 \vdash_{\mathsf{U}} \mathsf{e}_1 : \sigma \\ \Gamma, \mathsf{x}_2 : \sigma_2 \vdash_{\mathsf{U}} \mathsf{e}_2 : \sigma\end{array}}{\Gamma \vdash_{\mathsf{U}} \mathsf{case}\, \mathsf{e}\, \mathsf{of}\, \mathsf{x}_1.\, \mathsf{e}_1 \mid \mathsf{x}_2.\, \mathsf{e}_2 : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \sigma[\mu\alpha.\, \sigma/\alpha]}{\Gamma \vdash_{\mathsf{U}} \mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{e} : \mu\alpha.\, \sigma} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \mu\alpha.\, \sigma}{\Gamma \vdash_{\mathsf{U}} \mathsf{unfold}\, \mathsf{e} : \sigma[\mu\alpha.\, \sigma/\alpha]}$$

$$\frac{\Gamma, \alpha \vdash_{\mathsf{U}} \mathsf{v} : \sigma}{\Gamma \vdash_{\mathsf{U}} \Lambda\alpha.\, \mathsf{v} : \forall\alpha.\, \sigma} \qquad \frac{\Gamma \vdash_{\mathsf{U}} \mathsf{e} : \forall\alpha.\, \sigma \qquad \Gamma \vdash \sigma'}{\Gamma \vdash_{\mathsf{U}} \mathsf{e}\, [\sigma'] : \sigma[\sigma'/\alpha]}$$

Fig. 2. Unrestricted Language: Static Semantics

The novelty lies in the linear language λ^{L}, which we present in several steps. As is common in λ-calculi with references, the small-step operational semantics is given for a language that is not exactly the surface language in which programs are written, because memory allocation returns *locations* ℓ that are not in the grammar of surface terms. Reductions are defined on *configurations*, a local store paired with a term in a slightly larger *internal* language. We have two type systems, a type system on surface terms, that does not mention locations and stores—which is the one a programmer needs to know—and a type system on configurations, which contains enough static information to reason about the dynamics of our language and prove subject reduction. Again, this follows the standard structure of syntactic soundness proofs for languages with a mutable store.

We present the surface language and type system in Section 2.1, except for the language fragment manipulating the linear store which is presented in Section 2.2. Finally, the internal terms, their typing and reduction semantics are presented in Section 2.3.

2.1 The Core of λ^{L}

Figure 3 presents the surface syntax of our linear language λ^{L}. For the syntactic categories of types σ, and expressions e, the last line contains the constructions related to the linear store that we only discuss in Section 2.2.

Types	σ ::=	$\sigma_1 \otimes \sigma_2 \mid 1 \mid \sigma_1 \multimap \sigma_2 \mid \sigma_1 \oplus \sigma_2 \mid \mu\alpha.\,\sigma \mid \alpha \mid !\sigma \mid \mathsf{Box}\ 1\ \sigma \mid \mathsf{Box}\ 0$
Expr.	e ::=	$\mathsf{x} \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \mid \mathsf{let}\ \langle \mathsf{v}_1, \mathsf{v}_2 \rangle = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \langle\rangle \mid \mathsf{e}_1 ; \mathsf{e}_2 \mid \lambda(\mathsf{x} : \sigma).\,\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid$ $\mathsf{inj}_1\ \mathsf{e} \mid \mathsf{inj}_2\ \mathsf{e} \mid \mathsf{case}\ \mathsf{e}'\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 \mid \mathsf{fold}_{\mu\alpha.\sigma}\ \mathsf{e} \mid \mathsf{unfold}\ \mathsf{e} \mid$ $\mathsf{share}\ \mathsf{e} \mid \mathsf{copy}\ \mathsf{e} \mid \mathsf{new}\ \mathsf{e} \mid \mathsf{free}\ \mathsf{e} \mid \mathsf{box}\ \mathsf{e} \mid \mathsf{unbox}\ \mathsf{e}$
Values	v ::=	$\mathsf{x} \mid \langle \mathsf{v}_1, \mathsf{v}_2 \rangle \mid \langle\rangle \mid \lambda(\mathsf{x} : \sigma).\,\mathsf{e} \mid \mathsf{inj}_1\ \mathsf{v} \mid \mathsf{inj}_2\ \mathsf{v} \mid \mathsf{fold}_{\mu\alpha.\sigma}\ \mathsf{v} \mid \mathsf{share}\ \mathsf{v}$
Contexts	Γ ::=	$\cdot \mid \Gamma, \mathsf{x} : \sigma$

Fig. 3. Linear Language: Surface Syntax

In technical terms, our linear type system is exactly propositional intuitionistic linear logic, extended with iso-recursive types. For simplicity and because we did not need them, our current system also does not have polymorphism or additive/lazy pairs $\sigma_1 \,\&\, \sigma_2$. Additive pairs would be a trivial addition, but polymorphism would require more work when we define the multi-language semantics in Section 3.

In less technical terms, our type system can enforce that values be used *linearly*, meaning that they cannot be duplicated or erased, they have to be deconstructed exactly once. Only some types have this linearity restriction; others allow duplication and sharing of values at will. We can think of linear values as *resources* to be spent wisely; for any linear value somewhere in a term, there can be only one way to access this value, so we can interpret the language as enforcing an *ownership* discipline where whoever points to a linear value owns it.

The types of linear values are the type of linear pairs $\sigma_1 \otimes \sigma_2$, of linear disjoint unions $\sigma_1 \oplus \sigma_2$, of linear functions $\sigma_1 \multimap \sigma_2$, and of the linear unit type 1. For example, a linear function must be called exactly once, and its result must in turn be consumed – such linear functions can safely capture linear resources. The expression-formers at these types use the same syntax as the unrestricted language λ^{U}, with the exception of linear pair deconstruction $\mathsf{let}\ \langle \mathsf{v}_1, \mathsf{v}_2 \rangle = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2$, which names both members of the deconstructed pair at once. A linear pair type with projection would only ever allow to observe one of the two members; this would correspond to the additive/lazy pairs $\sigma_1 \,\&\, \sigma_2$, where only one of the two members is ever computed.

The types of non-linear, duplicable values are the types of the form $!\sigma$—the *exponential* modality of linear logic. If e has type σ, the term $\mathsf{share}\ \mathsf{e}$ has type $!\sigma$. Values of this type are not uniquely owned, they can be shared at will. If the term e has duplicable type $!\sigma$, then the term $\mathsf{copy}\ \mathsf{e}$ has type σ: this creates a local copy of the value that is uniquely-owned by its receiver and must be consumed linearly.

This resource-usage discipline is enforced by the surface typing rules of λ^{L}, presented in Figure 4. They are exactly the standard (two-sided) logical rules of intuitionistic linear logic, annotated with program terms. The non-duplicability of linear values is enforced by the way contexts are merged by the inference

$$\boxed{\Gamma_1 \curlyvee \Gamma_2}$$

$$
\begin{array}{lcll}
(\Gamma_1, \mathsf{x}{:}!\sigma) \curlyvee (\Gamma_2, \mathsf{x}{:}!\sigma) & \stackrel{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}!\sigma & \\
(\Gamma_1, \mathsf{x}{:}\sigma) \curlyvee \Gamma_2 & \stackrel{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}\sigma & (\mathsf{x} \notin \Gamma_2) \\
\Gamma_1 \curlyvee (\Gamma_2, \mathsf{x}{:}\sigma) & \stackrel{\text{def}}{=} & (\Gamma_1 \curlyvee \Gamma_2), \mathsf{x}{:}\sigma & (\mathsf{x} \notin \Gamma_1)
\end{array}
$$

$$\boxed{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma}$$

$$\frac{}{!\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{L}} \mathsf{x} : \sigma} \qquad \frac{\Gamma_1 \vdash_{\mathrm{L}} \mathsf{e}_1 : \sigma_1 \qquad \Gamma_2 \vdash_{\mathrm{L}} \mathsf{e}_2 : \sigma_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash_{\mathrm{L}} \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \sigma_1 \otimes \sigma_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_1 \otimes \sigma_2 \\ \Gamma', \mathsf{x}_1{:}\sigma_1, \mathsf{x}_2{:}\sigma_2 \vdash_{\mathrm{L}} \mathsf{e}' : \sigma\end{array}}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{let}\ \langle \mathsf{x}_1, \mathsf{x}_2 \rangle = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' : \sigma}$$

$$\frac{}{!\Gamma \vdash_{\mathrm{L}} \langle\rangle : \mathsf{1}} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \mathsf{1} \qquad \Gamma' \vdash_{\mathrm{L}} \mathsf{e}' : \sigma}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{e}; \mathsf{e}' : \sigma} \qquad \frac{\Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{L}} \mathsf{e} : \sigma'}{\Gamma \vdash_{\mathrm{L}} \lambda(\mathsf{x}:\sigma).\,\mathsf{e} : \sigma \multimap \sigma'}$$

$$\frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma' \multimap \sigma \qquad \Gamma' \vdash_{\mathrm{L}} \mathsf{e}' : \sigma'}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{e}\ \mathsf{e}' : \sigma} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_i}{\Gamma \vdash_{\mathrm{L}} \mathsf{inj}_{\mathsf{i}}\ \mathsf{e} : \sigma_1 \oplus \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma_1 \oplus \sigma_2 \qquad \begin{array}{c}\Gamma', \mathsf{x}_1 : \sigma_1 \vdash_{\mathrm{L}} \mathsf{e}_1 : \sigma \\ \Gamma', \mathsf{x}_2 : \sigma_2 \vdash_{\mathrm{L}} \mathsf{e}_2 : \sigma\end{array}}{\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} \mathsf{case}\ \mathsf{e}\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 : \sigma} \qquad \frac{!\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma}{!\Gamma \vdash_{\mathrm{L}} \mathsf{share}\ \mathsf{e} : !\sigma} \qquad \frac{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : !\sigma}{\Gamma \vdash_{\mathrm{L}} \mathsf{copy}\ \mathsf{e} : \sigma}$$

$$\mu\alpha.\,\sigma \underset{\mathsf{fold}_{\mu\alpha.\sigma}}{\overset{\mathsf{unfold}}{\rightleftarrows}} \sigma[\mu\alpha.\,\sigma/\alpha] \qquad \mathsf{1} \underset{\mathsf{free}}{\overset{\mathsf{new}}{\rightleftarrows}} \mathsf{Box}\ 0 \qquad \mathsf{Box}\ 1\ \sigma \underset{\mathsf{box}}{\overset{\mathsf{unbox}}{\rightleftarrows}} (\mathsf{Box}\ 0) \otimes \sigma$$

Fig. 4. Linear Language: Surface Static Semantics

rules: if e_1 is type-checked in the context Γ_1 and e_2 in Γ_2, then the linear pair $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ is only valid in the combined context $\Gamma_1 \curlyvee \Gamma_2$. The $(\curlyvee)$ operation is partial; this combined context is defined only if the variables shared by Γ_1 and Γ_2 are duplicable—their type is of the form $!\sigma$. In other words, a variable at a non-duplicable type in $\Gamma_1 \curlyvee \Gamma_2$ cannot possibly appear in both Γ_1 and Γ_2: it must appear exactly once[6].

A good way to think of the linear judgment $\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma$ is that the evaluation of e *consumes* the linear variables of Γ; it is thus natural that the strict pair $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ would need separate sets of resources Γ_1 and Γ_2, as it evaluates both members to return a value. On the other hand, case elimination $\mathsf{case}\,\mathsf{e}\,\mathsf{of}\,\mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2$ reuses the same context Γ' in both branches e_1 and e_2: only one will be evaluated, so they do not compete for resources.

The variable rule does not expect a context of the form $\Gamma, \mathsf{x}{:}\sigma$ but of the form $!\Gamma, \mathsf{x}{:}\sigma$. Here $!\Gamma$ is a notation for the pointwise application of the $(!)$ connective to all the types in Γ—i.e., all types in $!\Gamma$ are of the form $!\sigma$. This means that the variable rule can only be used when all variables in the context are duplicable, except maybe the variable that is being used. A context of the form $\Gamma, \mathsf{x}{:}\sigma$ would allow us to forget some variable present in the context; in our judgment $\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma$, all non-duplicable variables in Γ must appear (once) in e.

The form $!\Gamma$ is also used in the typing rule for $\mathsf{share}\,\mathsf{e}$: a term can only be made duplicable if it does not depend on linear resources from the context. Otherwise, duplicating the shared value could break the unique-ownership discipline on these linear resources.

Finally, the linear isomorphism notation for fold and unfold in Figure 4 defines them as primitive functions, at the given linear function type, in the empty context – using them does not consume resources. This notation also means that, operationally, these two operations shall be inverses of each other. The rules for the linear store type $\mathsf{Box}\ 1\ \sigma$ and $\mathsf{Box}\ 0$ are described in Section 2.2.

Lemma 1 (Context joining properties) *Context joining* $(\curlyvee)$ *is partial but associative and commutative. In particular, if* $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee \Gamma'$ *is defined, then both* $\Gamma_i \curlyvee \Gamma'$ *are defined.*

2.2 Linear Memory in λ^{L}

The surface typing rules for the linear store are given at the end of Figure 4. The linear type $\mathsf{Box}\ 1\ \sigma$ represents a memory location that holds a value of type σ. The type $\mathsf{Box}\ 0$ represents a location that has been allocated, but does not currently hold a value. The primitive operations to act on this type are given as linear isomorphisms: new allocates, turning a unit value into an empty location; conversely, free reclaims an empty location. Putting a value into the location and taking it out are expressed by box and unbox, which convert between a pair of an empty location and a value, of type $(\mathsf{Box}\ 0) \otimes \sigma$, and a full location, of type $\mathsf{Box}\ 1\ \sigma$.

[6] Standard presentations of linear logic force contexts to be completely distinct, but have a separate rule to duplicate linear variables, which is less natural for programming.

For example, the following program takes a full reference and a value, and swaps the value with the content of the reference:

$$\lambda(\mathsf{p} : (\mathsf{Box}\ 1\ \sigma) \otimes \sigma).\ \mathsf{let}\ \langle \mathsf{r}, \mathsf{x} \rangle = \mathsf{p}\ \mathsf{in}\ \mathsf{let}\ \langle \mathsf{l}, \mathsf{x}_l \rangle = \mathsf{unbox}\ \mathsf{r}\ \mathsf{in}\ \langle \mathsf{box}\ \langle \mathsf{l}, \mathsf{x} \rangle, \mathsf{x}_l \rangle$$

The programming style following from this presentation of linear memory is functional, or applicative, rather than imperative. Rather than insisting on the mutability of references—which is allowed by the linear discipline—we may think of the type $\mathsf{Box}\ 1\ \sigma$ as representing the indirection through the heap that is implicit in functional programs. In a sense, we are not writing imperative programs with a mutable store, but rather making explicit the allocations and dereferences happening in higher-level purely functional language. In this view, empty cells allow memory reuse.

This view that $\mathsf{Box}\ 1\ \sigma$ represents indirection through the memory suggests we can encode lists of values of type σ by the type $\mathsf{LinList}\ \sigma \stackrel{\text{def}}{=} \mu\alpha.\ 1 \oplus \mathsf{Box}\ 1\ (\sigma \otimes \alpha)$. The placement of the box inside the sum mirrors the fact that empty list is represented as an immediate value in functional languages. From this type definition, one can write an in-place reverse function on lists of σ as follows:

$$\begin{array}{l}
\mathsf{fix}\ \lambda(\mathsf{rev_into} : \mathsf{LinList}\ \sigma \multimap \mathsf{LinList}\ \sigma \multimap \mathsf{LinList}\ \sigma). \\
\qquad \lambda(\mathsf{xs} : \mathsf{LinList}\ \sigma).\ \lambda(\mathsf{acc} : \mathsf{LinList}\ \sigma). \\
\quad \mathsf{case\ unfold\ xs\ of} \\
\quad \mid \mathsf{y}.\ (\mathsf{y}; \mathsf{acc}) \\
\quad \mid \mathsf{y}.\ \mathsf{let}\ \langle \mathsf{l}, \mathsf{p} \rangle = \mathsf{unbox}\ \mathsf{y}\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ \langle \mathsf{xs}, \mathsf{x} \rangle = \mathsf{p}\ \mathsf{in} \\
\qquad\quad \mathsf{rev_into}\ \mathsf{xs}\ (\mathsf{fold}\ (\mathsf{inj}_2\ (\mathsf{box}\ \langle \mathsf{l}, \langle \mathsf{x}, \mathsf{acc} \rangle \rangle)))
\end{array}$$

This definition uses a fixpoint operator fix that can be defined, in the standard way, using the iso-recursive type $\mu\alpha.\ \alpha \multimap \sigma \multimap \sigma'$ of the strict fixpoint combinator on functions $\sigma \multimap \sigma'$.

Our linear language λ^{L} is a formal language that is not terribly convenient to program directly. We will not present a full surface language in this work, but one could easily define syntactic sugar to write the exact same function as follows:

$$\begin{array}{ll}
\mathsf{rev_into}\ \mathsf{Nil} & \mathsf{acc} = \mathsf{acc} \\
\mathsf{rev_into}\ (\mathsf{Cons}\ \langle \mathsf{x}, \mathsf{xs} \rangle @ \mathsf{l}) & \mathsf{acc} = \mathsf{rev_into}\ \mathsf{xs}\ (\mathsf{Cons}\ \langle \mathsf{x}, \mathsf{acc} \rangle @ \mathsf{l})
\end{array}$$

One can read this function as the usual functional `rev_append` function on lists, annotated with memory reuse information: if we assume we are the unique owner of the input list and won't need it anymore, we can reuse the memory of its cons cells (given in this example the name l) to store the reversed list. On the other hand, if you read the box and unbox as imperative operations, this code expresses the usual imperative pointer-reversal algorithm.

This double view of linear state occurs in other programming systems with linear state. It was recently emphasized in O'Connor, Chen, Rizkallah, Amani, Lim, Murray, Nagashima, Sewell, and Klein [2016], where the functional point of view is seen as easing formal verification, while the imperative view is used as a compilation technique to produce efficient C code from linear programs.

2.3 Internal λ^{L} Syntax and Typing

To give a dynamic semantics for λ^{L} and prove it sound, we need to extend the language with explicit stores and store locations. Indeed, the allocating term $\mathsf{new}\ \langle\rangle$ should reduce to a "fresh location" ℓ allocated in some store s, and neither are part of the surface-language syntax. The corresponding internal typing judgment is more complex, but note that users do not need to know about it to reason about correctness of surface programs. The internal typing is essential for the soundness proof, but also useful for defining the multi-language semantics in Section 3.

The syntax of internal terms and the internal type system are presented in Figure 5. Reduction will be defined on *configurations* $(s \mid \mathsf{e})$, which are pairs of a store s and a term e. Stores s map *locations* ℓ to either nothing (the location is empty), written $[\ell \mapsto \cdot]$, or a value paired with its own local store, noted $[\ell \mapsto (s \mid \mathsf{v})]$. Having local stores in this way, instead of a single global store as is typical in formalizations of ML, directly expresses the idea of "memory ownership" in the syntax: a term e "owns" the locations that appear in it, and a configuration $(s \mid \mathsf{e})$ is only well-typed if the domain of s is exactly those locations. Each store slot, in turn, may contain a value and the local store owned by the value; in particular, passing a full location of type $\mathsf{Box}\ 1\ \sigma$ transfers ownership of the location, but also of the store fragment captured by the value.

Our internal typing judgment $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ checks configurations, not just terms, and relies not only on a typing context for variables Γ but also on a *store typing* Ψ, which maps the locations of the configuration to typing assumptions of two forms: $(\cdot; \cdot \vdash \ell : \mathsf{Box}\ 0)$ indicates that ℓ must be empty in the configuration, and $(\Gamma; \Psi \vdash \ell : \mathsf{Box}\ 1\ \sigma)$ indicates that ℓ is full, and that the value it contains owns a local store of type s and the resources in Γ.

Just as linear variables must occur exactly once in a term, locations have linear types and thus occur exactly once in a term. Our typing judgment uses disjoint store typings $\Psi_1 \uplus \Psi_2$ to enforce this linearity. Similarly, leaf rules such as the variable, unit, and location rules enforce that both the store typing and the store be empty, which enforces that all locations are used in the term.

Locations ℓ are always linear, never duplicable. To allow sharing terms that contain locations, the internal language uses the internal construction $\mathsf{share}(s : \Psi).\ \mathsf{e}$, that *captures* a local store $s : \Psi$. This notation is a binding construct: the locations in s are bound by this shared term, and not visible outside this term. In particular, the typing rule for $\mathsf{share}(s : \Psi).\ \mathsf{e}$ checks the term e in the store s, but it is itself only valid paired with an empty store, under the empty store typing. When new copies of a shared term are made, the local store is copied as well: this is necessary to guarantee that locations remain linear—and for correctness of linear state update.

The typing rule for functions $\lambda(\mathsf{x} : \sigma).\ \mathsf{e}$ lets function bodies use an arbitrary store typing Ψ. This would be unsound if our functions were duplicable, but it is a natural and expressive choice for linear, one-shot functions. To make a function duplicable, one can share it at type $!(\sigma \multimap \sigma')$, whose values are of the canonical

$$
\begin{array}{llll}
\textit{Types} & \sigma & & \text{(unchanged from Figure 4)}\\
\textit{Expressions} & \mathsf{e} & +::= & \cdots \mid \ell \mid \mathsf{share}(s:\Psi).\,\mathsf{e}\\
& & & \text{with } \mathsf{share}\,\mathsf{e} \overset{\text{def}}{=} \mathsf{share}(\emptyset:\cdot).\,\mathsf{e}\\
\textit{Values} & \mathsf{v} & +::= & \cdots \mid \ell \mid \mathsf{share}(s:\Psi).\,\mathsf{v}\\
\textit{Store} & s & ::= & \emptyset \mid s[\ell \mapsto (s \mid \mathsf{v})] \mid s[\ell \mapsto \cdot]\\
\textit{Configurations} & & ::= & (s \mid \mathsf{e})\\
\textit{Store typing} & \Psi & ::= & \cdot \mid \Psi, (\cdot;\cdot \vdash \ell : \mathsf{Box}\ 0)\\
& & & \mid \Psi, (\Psi';\Gamma \vdash \ell : \mathsf{Box}\ 1\ \sigma)
\end{array}
$$

$\boxed{s_1 +\!\!\!+\, s_2}$ Union of stores on disjoint locations $\boxed{\Psi_1 \uplus \Psi_2}$ Union of store typings on disjoint locations

$\boxed{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma}$ $\boxed{\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma \overset{\text{def}}{=} \cdot;\Gamma \vdash_{\mathrm{L}} \emptyset \mid \mathsf{e} : \sigma}$

$$\frac{\Psi_1;\Gamma_1 \vdash_{\mathrm{L}} s_1 \mid \mathsf{e}_1 : \sigma_1 \qquad \Psi_2;\Gamma_2 \vdash_{\mathrm{L}} s_2 \mid \mathsf{e}_2 : \sigma_2}{\Psi_1 \uplus \Psi_2;\Gamma_1 \curlyvee \Gamma_2 \vdash_{\mathrm{L}} s_1 +\!\!\!+\, s_2 \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \sigma_1 \otimes \sigma_2}$$

$$\frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma_1 \otimes \sigma_2 \qquad \Psi';\Gamma', \mathsf{x}_1 : \sigma_1, \mathsf{x}_2 : \sigma_2 \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma}{\Psi \uplus \Psi';\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} s +\!\!\!+\, s' \mid \mathsf{let}\ \langle \mathsf{x}_1, \mathsf{x}_2 \rangle = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' : \sigma}$$

$$\frac{}{\cdot;!\Gamma, \mathsf{x}:\sigma \vdash_{\mathrm{L}} \emptyset \mid \mathsf{x} : \sigma} \qquad \frac{}{\cdot;!\Gamma \vdash_{\mathrm{L}} \emptyset \mid \langle\rangle : 1} \qquad \frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : 1 \qquad \Psi';\Gamma' \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma}{\Psi \uplus \Psi';\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} s +\!\!\!+\, s' \mid \mathsf{e};\mathsf{e}' : \sigma}$$

$$\frac{\Psi;\Gamma, \mathsf{x}:\sigma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma'}{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e} : \sigma \multimap \sigma'} \qquad \frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma' \multimap \sigma \qquad \Psi';\Gamma' \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma'}{\Psi \uplus \Psi';\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} s +\!\!\!+\, s' \mid \mathsf{e}\ \mathsf{e}' : \sigma}$$

$$\frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma_i}{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{inj_i}\,\mathsf{e} : \sigma_1 \oplus \sigma_2}$$

$$\frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma_1 \oplus \sigma_2 \qquad \Psi';\Gamma', \mathsf{x}_1 : \sigma_1 \vdash_{\mathrm{L}} s' \mid \mathsf{e}_1 : \sigma \qquad \Psi';\Gamma', \mathsf{x}_2 : \sigma_2 \vdash_{\mathrm{L}} s' \mid \mathsf{e}_2 : \sigma}{\Psi \uplus \Psi';\Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} s +\!\!\!+\, s' \mid \mathsf{case}\ \mathsf{e}\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 : \sigma}$$

$$\frac{\Psi;!\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma}{\cdot;!\Gamma \vdash_{\mathrm{L}} \emptyset \mid \mathsf{share}(s:\Psi).\,\mathsf{e} : !\sigma} \qquad \frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : !\sigma}{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{copy}\ \mathsf{e} : \sigma}$$

$$\frac{}{(\cdot;\cdot \vdash \ell : \mathsf{Box}\ 0);!\Gamma \vdash_{\mathrm{L}} [\ell \mapsto \cdot] \mid \ell : \mathsf{Box}\ 0}$$

$$\frac{\Psi;\Gamma \vdash_{\mathrm{L}} s \mid \mathsf{v} : \sigma}{(\Psi;\Gamma \vdash \ell : \mathsf{Box}\ 1\ \sigma);\Gamma \curlyvee !\Gamma' \vdash_{\mathrm{L}} [\ell \mapsto (s \mid \mathsf{v})] \mid \ell : \mathsf{Box}\ 1\ \sigma}$$

$$\mu\alpha.\,\sigma \underset{\mathsf{fold}_{\mu\alpha.\sigma}}{\overset{\mathsf{unfold}}{\rightleftarrows}} \sigma[\mu\alpha.\,\sigma/\alpha] \qquad 1 \underset{\mathsf{free}}{\overset{\mathsf{new}}{\rightleftarrows}} \mathsf{Box}\ 0 \qquad \mathsf{Box}\ 1\ \sigma \underset{\mathsf{box}}{\overset{\mathsf{unbox}}{\rightleftarrows}} (\mathsf{Box}\ 0) \otimes \sigma$$

Fig. 5. Linear Language: Internal Static Semantics

form $\mathsf{share}(s : \Psi).\, \lambda(\mathsf{x} : \sigma).\, \mathsf{e}$. It is the sharing construct, not the function itself, that closes over the local store.

With the macro-expansion $\mathsf{share}\,\mathsf{e} \stackrel{\text{def}}{=} \mathsf{share}(\emptyset : \cdot).\, \mathsf{e}$, any term e of the surface language (Figure 3) can be seen as a term of the internal language (Figure 5). In particular, we can prove that the surface and internal typing judgments coincide on surface terms.

Lemma 2 *If* e *is a surface term of* λ^{L}*, then the surface judgment* $\Gamma \vdash_{\mathrm{L}} \mathsf{e} : \sigma$ *holds if and only if the internal judgment* $\cdot; \Gamma \vdash_{\mathrm{L}} \emptyset \mid \mathsf{e} : \sigma$ *holds.*

The following technical results are used in the soundness proof for the language – the subject-reduction result.

Lemma 3 (Inversion principle for λ^{L} values) *In any complete derivation of* $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{v} : \sigma$*, either* v *is a variable* x*, or the derivation starts with the introduction rule for* σ*.*

For example, if we have $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{v} : !\sigma$, then we know that v is either a variable or of the form $\mathsf{share}(s' : \Psi').\, \mathsf{v}'$ for some v', but also that $s = \emptyset$, $\Psi = \cdot$ and that Γ is of the form $!\Gamma'$ for some Γ'. The latter is immediate if v is $\mathsf{share}(s' : \Psi').\, \mathsf{v}'$, and also holds if v is a variable.

Lemma 4 (Weakening by duplicable contexts) $\Psi; \Gamma' \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ *implies* $\Psi; !\Gamma, \Gamma' \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$*.*

2.4 Reduction of Internal Terms

Figure 6 gives a small-step operational semantics for the internal terms of λ^{L}. We separate the head reductions ($\stackrel{\mathsf{L}}{\rightsquigarrow}$) from reductions in depth ($\stackrel{\mathsf{L}}{\hookrightarrow}$). The head reduction of the linear types of the core language do not involve the store and are standard. For the store primitives of Figure 4 acting on $\mathsf{Box}\ 0\,, \mathsf{Box}\ 1\ \sigma$, we reuse the isomorphism notation to emphasize that the related primitives are inverses of each other.

There are several reduction rules for copy ($\mathsf{share}\,\mathsf{e}$), one for each type connective. These reductions perform a deep copy of the value, stopping only on ground data ($\langle\rangle$), function values, and shared sub-terms: when copying a $!!\sigma$ into a $!\sigma$, there is no need for a deep copy. When it encounters a location, copy ($\mathsf{share}\,\ell$) reduces to a new allocation. If the location contains a value, the new location is filled with a copy of this value.

The copying rule for functions performs a copy of the local store s of the shared function. The locations in s are bound on the left-hand-side of the reduction, and free on the right-hand-side: this reduction step allocates fresh locations, and the store typing of the term changes from $\cdot$ on the left to Ψ on the right. The fact that reduction changes the store typing is not unique to this rule, it is also the case when directly copying locations. In ML languages with references, the store only grows during reduction. That is not the case for our linear store: our reduction may either allocate new locations or free existing ones.

head reduction $\boxed{\mathsf{e} \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{e}'}$ $\boxed{(s \mid \mathsf{e}) \overset{\mathsf{L}}{\rightsquigarrow} (s' \mid \mathsf{e}')}$

$$\begin{array}{rcl}
\mathsf{let}\ \langle \mathsf{x}_1, \mathsf{x}_2 \rangle = \langle \mathsf{v}_1, \mathsf{v}_2 \rangle\ \mathsf{in}\ \mathsf{e} & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{e}[\mathsf{v}_1/\mathsf{x}_1][\mathsf{v}_2/\mathsf{x}_2] \\
\langle\rangle; \mathsf{e} & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{e} \\
(\lambda(\mathsf{x}:\sigma).\,\mathsf{e})\ \mathsf{v} & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{e}[\mathsf{v}/\mathsf{x}] \\
\mathsf{case}\ (\mathsf{inj}_i\ \mathsf{v})\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{e}_i[\mathsf{v}/\mathsf{x}_i] \\
\mathsf{unfold}\ (\mathsf{fold}_{\mu\alpha.\sigma}\ \mathsf{v}) & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{v}
\end{array}
\qquad
\frac{\mathsf{e} \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{e}'}{(s \mid \mathsf{e}) \overset{\mathsf{L}}{\rightsquigarrow} (s \mid \mathsf{e}')}$$

$$(\emptyset \mid \langle\rangle) \underset{\overset{\mathsf{L}}{\leftsquigarrow}\ \mathsf{free}}{\overset{\mathsf{new}\ \overset{\mathsf{L}}{\rightsquigarrow}}{}} ([\ell \mapsto \cdot] \mid \ell) \qquad (s[\ell \mapsto \cdot] \mid \langle \ell, \mathsf{v} \rangle) \underset{\overset{\mathsf{L}}{\leftsquigarrow}\ \mathsf{unbox}}{\overset{\mathsf{box}\ \overset{\mathsf{L}}{\rightsquigarrow}}{}} ([\ell \mapsto (s \mid \mathsf{v})] \mid \ell)$$

$$\begin{array}{l}
\mathsf{copy}\ (\mathsf{share}(s_1 \mathbin{+\!\!+} s_2 : \Psi_1 \uplus \Psi_2).\,\langle \mathsf{v}_1, \mathsf{v}_2 \rangle) \\
\overset{\mathsf{L}}{\rightsquigarrow} \qquad \text{if } \mathsf{locs}(s_i) = \mathsf{locs}(\Psi_i) = \mathsf{locs}(\mathsf{v}_i) \\
\langle \mathsf{copy}\ \mathsf{share}(s_1 : \Psi_1).\,\mathsf{v}_1, \mathsf{copy}\ \mathsf{share}(s_2 : \Psi_2).\,\mathsf{v}_2 \rangle
\end{array}$$

$$\begin{array}{lcl}
\mathsf{copy}\ (\mathsf{share}(\emptyset : \cdot).\,\langle\rangle) & \overset{\mathsf{L}}{\rightsquigarrow} & \langle\rangle \\
\mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\mathsf{inj}_i\ \mathsf{v}) & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{inj}_i\ \mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\mathsf{v}) \\
\mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\mathsf{fold}\ \mathsf{v}) & \overset{\mathsf{L}}{\rightsquigarrow} & \mathsf{fold}\ (\mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\mathsf{v}))
\end{array}$$

$$(\emptyset \mid \mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\lambda(\mathsf{x}:\sigma).\,\mathsf{e})) \overset{\mathsf{L}}{\rightsquigarrow} (s \mid \lambda(\mathsf{x}:\sigma).\,\mathsf{e})$$

$$\mathsf{copy}\ (\mathsf{share}(\emptyset : \cdot).\,(\mathsf{share}(s : \Psi).\,\mathsf{v})) \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{share}(s : \Psi).\,\mathsf{v}$$

$$\mathsf{copy}\ (\mathsf{share}([\ell \mapsto \cdot] : (\cdot; \cdot \vdash \ell : \mathsf{Box}\ 0)).\,\ell) \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{new}\ \langle\rangle$$

$$\begin{array}{l}
\mathsf{copy}\ (\mathsf{share}([\ell \mapsto (s \mid \mathsf{v})] : (\Psi; !\Gamma \vdash \ell : \mathsf{Box}\ 1\ \sigma)).\,\ell) \\
\qquad \overset{\mathsf{L}}{\rightsquigarrow} \mathsf{box}\ \langle \mathsf{new}\ \langle\rangle, \mathsf{copy}\ (\mathsf{share}(s : \Psi).\,\mathsf{v}) \rangle
\end{array}$$

linear reduction contexts $\boxed{\Psi; \Gamma \vdash_{\mathrm{L}} s \mid K[\Box{:}\sigma] : \sigma'}$

$$\begin{array}{rl}
K ::= & \Box{:}\sigma \mid \langle K, \mathsf{e}_2 \rangle \mid \langle \mathsf{v}, K \rangle \mid \mathsf{let}\ \langle \mathsf{v}_1, \mathsf{v}_2 \rangle = K\ \mathsf{in}\ \mathsf{e}_2 \mid \\
& K; \mathsf{e} \mid K\ \mathsf{e} \mid \mathsf{v}\ K \mid \mathsf{copy}\ K \mid \\
& \mathsf{inj}_1\ K \mid \mathsf{inj}_2\ K \mid \mathsf{case}\ K\ \mathsf{of}\ \mathsf{x}_1.\,\mathsf{e}_1 \mid \mathsf{x}_2.\,\mathsf{e}_2 \mid \\
& \mathsf{fold}_{\mu\alpha.\sigma}\ K \mid \mathsf{unfold}\ K \mid \\
& \mathsf{new}\ K \mid \mathsf{free}\ K \mid \mathsf{box}\ K \mid \mathsf{unbox}\ K
\end{array}$$

typing rules of terms, plus: $\dfrac{}{\cdot; \cdot \vdash_{\mathrm{L}} \emptyset \mid (\Box{:}\sigma) : \sigma}$

reduction $\boxed{(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')}$

$$\frac{(s \mid \mathsf{e}) \overset{\mathsf{L}}{\rightsquigarrow} (s' \mid \mathsf{e}')}{(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')} \qquad \frac{\Psi; \Gamma \vdash_{\mathrm{L}} s'' \mid K[\Box{:}\sigma] : \sigma' \quad (s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')}{(s'' \mathbin{+\!\!+} s \mid K[\mathsf{e}]) \overset{\mathsf{L}}{\hookrightarrow} (s'' \mathbin{+\!\!+} s' \mid K[\mathsf{e}'])}$$

$$\frac{\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma \quad (s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}') \quad \Psi'; \Gamma \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma}{(\emptyset \mid \mathsf{share}(s : \Psi).\,\mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (\emptyset \mid \mathsf{share}(s' : \Psi').\,\mathsf{e}')}$$

Fig. 6. Linear Language: Operational Semantics

We define a grammar of (deterministic) reduction contexts, which contain exactly one hole $\square$ in evaluation position. However, we only define *linear* contexts K that do not share their hole: we need a specific treatment of the $\mathsf{share}(s : \Psi).\,\mathsf{e}$ reduction. Its subterm e is reduced in the local store s, but may create or free locations in the store; so we need to update the local store and its store typing during the reduction.

Theorem 1 (Progress) *If $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$, then either e is a value v or there exists $(s' \mid \mathsf{e}')$ such that $(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')$.*

Proof. By induction on the typing derivation of e, using induction hypothesis in the evaluation order corresponding to the structure of contexts K. If one induction hypothesis returns a reduction, we build a bigger reduction ($\overset{\mathsf{L}}{\hookrightarrow}$) for the whole term. If all induction hypotheses return a value, the proof depends on whether the head term-former is an introduction/construction form or an elimination/destruction form. An introduction form whose subterms are values is a value. For elimination forms, we use Lemma 3 (Inversion principle for λ^{L} values) on the eliminated subterm (a value), to learn that it starts with an introduction form, and thus forms a head redex with the head elimination form, so we build a head reduction ($\overset{\mathsf{L}}{\rightsquigarrow}$).

Lemma 5 (Non-store-escaping substitution principle) *If*

$$\Psi'; \Gamma, \mathsf{x}{:}\sigma \vdash_{\mathrm{L}} s' \mid \mathsf{e} : \sigma' \qquad \Psi; \Gamma' \vdash_{\mathrm{L}} s \mid \mathsf{v} : \sigma \qquad \Gamma \curlyvee \Gamma' \qquad \mathsf{x} \notin \Psi$$

then

$$\Psi \uplus \Psi'; \Gamma \curlyvee \Gamma' \vdash_{\mathrm{L}} s +\!\!+ s' \mid \mathsf{e}[\mathsf{v}/\mathsf{x}] : \sigma'$$

Proof. The proof, summarized below, proceeds by induction on the typing derivation of e.

Most cases need an additional case analysis on whether the substituted type σ is a duplicable type of the form $!\sigma''$, as it influence whether it may appear in zero or several subterms of e. (This is a price to pay for contraction and weakening happening in all rules for convenience, instead of being isolated in separate structural rules.)

For example, in the variable case, e may be the variable x itself, in which case we know that Γ is empty and conclude immediately. But e may also be another variable y if x is duplicable and has been dropped. In that case, we perform an inversion (Lemma 3) on the v premise to learn that Ψ is empty and Γ' is duplicable, and can thus use Lemma 4 (Weakening by duplicable contexts).

In the $\langle \mathsf{e}_1, \mathsf{e}_2 \rangle$ case, if x is a linear variable it only occurs in one subterm on which we apply our induction hypothesis. If x is duplicable, inversion on the v premises again tells us that Γ' is duplicable. We know by assumption that $(\Gamma_1 \curlyvee \Gamma_2) \curlyvee \Gamma'$; because Γ' is duplicable, we can deduce from Lemma 1 (Context joining properties) that the $\Gamma_i \curlyvee \Gamma'$ are also defined, which let us apply an induction hypothesis on both subterms e_i. To conclude, we need the computation

$$\begin{aligned} & (\Gamma_1 \curlyvee \Gamma') \curlyvee (\Gamma_2 \curlyvee \Gamma') \\ = {} & \Gamma_1 \curlyvee \Gamma_2 \curlyvee (\Gamma' \curlyvee \Gamma') \\ = {} & \Gamma_1 \curlyvee \Gamma_2 \curlyvee \Gamma' \end{aligned}$$

which again comes from duplicability of Γ'.

The assumption $\mathsf{x} \notin \Psi$ enforces that the resource x is consumed in the term e itself, not in one of the values $[\ell \mapsto (s \mid \mathsf{v})]$ in the store: otherwise x would appear in the store typing $(\Gamma; \ell \vdash \Psi : \mathsf{Box}\ 1\ )$ of this location in Ψ. It is used in the case where $\mathsf{e} : \sigma$ is a full location $\ell : \mathsf{Box}\ 1\ \sigma'$. If x could appear in the value of ℓ in the store, we would have substitute it in the store as well – in our substitution statement, only the term is modified. Here we know that this value is unused, so it has a duplicable type and we can perform an inversion in the other cases.

Lemma 6 (Context decomposition) *If $\Psi'; \Gamma' \vdash_{\mathrm{L}} s' \mid K[\square{:}\sigma] : \sigma'$ holds, then $\Psi''; \Gamma'' \vdash_{\mathrm{L}} s'' \mid K[\mathsf{e}] : \sigma'$ holds if and only if there exists Ψ, Γ, s such that $\Psi'' = \Psi \uplus \Psi'$, $\Gamma'' = \Gamma \curlyvee \Gamma'$, $s'' = s \mathbin{+\!\!+} s'$ and $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$.*

Theorem 2 (Subject reduction for λ^{L}) *If $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ and $(s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}')$, then there exists a (unique) Ψ' such that $\Psi'; \Gamma \vdash_{\mathrm{L}} s' \mid \mathsf{e}' : \sigma$.*

Proof. The proof is done by induction on the reduction derivation.

The head-reduction rules involving substitutions rely on Lemma 5 (Non-store-escaping substitution principle); note that in each of them, for example $(\lambda(\mathsf{x}{:}\sigma').\,\mathsf{e}')\ \mathsf{e}''$, the substituted variable x is bound in the term e, and thus does not appear in the store s: the non-store-escaping hypothesis holds.

For the copy rule and the store operators, we build a valid derivation for the reduced configuration by inverting the typing derivation of the reducible configuration.

In the non-head-reduction cases, the share case is by direction, and the context case $K[\mathsf{e}]$ uses Lemma 6 (Context decomposition) to obtain a typing derivation for e, and the same lemma again rebuild a derivation of the reduced term $K[\mathsf{e}']$.

3 Multi-language semantics

To formally define our multi-language semantics we create a combined language λ^{UL} which lets us compose term fragments from both λ^{U} and λ^{L} together, and we give an operational semantics to this combined language. Interoperability is enabled by specifying how to transport values across the language boundaries.

Multi-language systems in the wild are not defined in this way: both languages are given a semantics, by interpretation or compilation, in terms of a shared lower-level language (C, assembly, the JVM or CLR bytecode, or Racket's core forms), and the two languages are combined at that level. Our formal multi-language description can be seen as a model of such combinations, that gives a specification of the expected observable behavior of this language combination.

Another difference from multi-languages in the wild is our use of very fine-grained language boundaries: a term written in one language can have its subterms written in the other, provided the type-checking rules allow it. Most multi-language systems, typically using Foreign Function Interfaces, offer coarser-grained composition at the level of compilation units. Fine-grained composition of existing languages, as done in the Eco project [Barrett, Bolz, Diekmann, and Tratt, 2016], is difficult because of semantic mismatches. In Section 5 (Hybrid program examples) we demonstrate that fine-grained composition is a rewarding language design, enabling new programming patterns.

3.1 Lump Type and Language Boundaries

The core components the multi-language semantics are shown Figure 7—the communication of values from one language to the other will be described in the next section. The multi-language λ^{UL} has two distinct syntactic categories of types, values, and expressions: those that come from λ^{U} and those that come from λ^{L}. Contexts, on the other hand, are mixed, and can have variables of both sorts. For a mixed context Γ, the notation $!\Gamma$ only applies ($!$) to its linear variables.

The typing rules of λ^{U} and λ^{L} are imported into our multi-language system, working on those two separate categories of program. They need to be extended to handle mixed contexts Γ instead of their original contexts Γ and Γ. In the linear case, the rules look exactly the same. In the ML case, the typing rules implicitly duplicate all the variables in the context. It would be unsound to extend them to arbitrary linear variables, so they use a duplicable context $!\Gamma$.

To build interesting multi-language programs, we need a way to insert a fragment coming from a language into a term written in another. This is done using *language boundaries*, two new term formers $\mathcal{LU}(\mathsf{e})$ and $\mathcal{UL}(s{:}\Psi \mid \mathsf{e})$ that inject an ML term into the syntactic category of linear terms, and a linear configuration into the syntactic category of ML terms.

Of course, we need new typing rules for these term-level constructions, clarifying when it is valid to send a value from λ^{U} into λ^{L} and vice versa. It would be incorrect to allow sending any type from one language into the other—for instance, by adding the counterpart of our language boundaries in the syntax of types—since values of linear types must be uniquely owned so they cannot possibly be sent to the ML side as the ML type system cannot enforce unique ownership.

On the other hand, any ML value could safely be sent to the linear world. For closed types, we could provide a corresponding linear type ($\mathsf{1}$ maps to $!\mathsf{1}$, etc.), but an ML value may also be typed by an abstract type variable α, in which case we can't know what the linear counterpart should be. Instead of trying to provide translations, we will send any ML type σ to the *lump type* $[\sigma]$, which embeds ML types into linear types. A lump is a blackbox, not a type translation: the linear language does not assume anything about the behavior of its values—the values of $[\sigma]$ are of the form $[\mathsf{v}]$, where $\mathsf{v} : \sigma$ is an ML value that the linear world cannot use. More precisely, we only propagate the information that ML values are all duplicable by sending σ to $![\sigma]$.

Types $\sigma \mid \sigma$

$$\sigma \qquad \text{(unchanged from Figure 1)}$$
$$\sigma \quad +{::=} \cdots \mid [\sigma]$$

Values $\mathsf{v} \mid \mathsf{v}$

$$\mathsf{v} \qquad \text{(unchanged from Figure 1)}$$
$$\mathsf{v} \quad +{::=} \cdots \mid [\mathsf{v}]$$

Expressions $\mathsf{e} \mid \mathsf{e}$

$$\mathsf{e} \quad +{::=} \cdots \mid \mathcal{UL}(s{:}\Psi \mid \mathsf{e})$$
$$\text{with} \quad \mathcal{UL}(\mathsf{e}) \overset{\text{def}}{=} \mathcal{UL}(\emptyset{:}\cdot \mid \mathsf{e})$$
$$\mathsf{e} \quad +{::=} \cdots \mid \mathcal{LU}(\mathsf{e})$$

Contexts $\Gamma ::= \cdot \mid \Gamma, \mathsf{x}{:}\sigma \mid \Gamma, \alpha \mid \Gamma, \mathsf{x}{:}\sigma$

Typing rules $\boxed{\Gamma \vdash_{\text{LU}} \mathsf{e} : \sigma}$ $\boxed{\Psi \mid \Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : \sigma}$

with $\Gamma \vdash_{\text{UL}} \mathsf{e} : \sigma \overset{\text{def}}{=} \cdot \mid \Gamma \vdash_{\text{UL}} \emptyset \mid \mathsf{e} : \sigma$

(Typing rules of $\Gamma \vdash_{\text{U}} \mathsf{e} : \sigma$ reused, with mixed context $!\Gamma$)
(Typing rules of $\Psi; \Gamma \vdash_{\text{L}} s \mid \mathsf{e} : \sigma$ reused, with mixed context Γ)

$$\frac{!\Gamma \vdash_{\text{LU}} \mathsf{e} : \sigma}{\cdot \mid !\Gamma \vdash_{\text{UL}} \emptyset \mid \mathcal{LU}(\mathsf{e}) : ![\sigma]} \qquad \frac{\Psi \mid !\Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : ![\sigma]}{!\Gamma \vdash_{\text{LU}} \mathcal{UL}(s{:}\Psi \mid \mathsf{e}) : \sigma}$$

Reduction rules

(Reduction rules of λ^{U} and λ^{L} reused unchanged) $\quad \dfrac{\mathsf{e} \overset{\mathsf{U}}{\hookrightarrow} \mathsf{e}'}{\mathcal{LU}(\mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} \mathcal{LU}(\mathsf{e}')}$

$$\frac{}{\mathcal{LU}(\mathsf{v}) \overset{\mathsf{L}}{\rightsquigarrow} [\mathsf{v}]} \qquad \frac{}{\mathcal{UL}(\emptyset{:}\cdot \mid \mathsf{share}\,[\mathsf{v}]) \overset{\mathsf{U}}{\hookrightarrow} \mathsf{v}}$$

$$\frac{\Psi \mid \Gamma \vdash_{\text{UL}} s \mid \mathsf{e} : \sigma \qquad (s \mid \mathsf{e}) \overset{\mathsf{L}}{\hookrightarrow} (s' \mid \mathsf{e}') \qquad \Psi' \mid \Gamma \vdash_{\text{UL}} s' \mid \mathsf{e}' : \sigma}{\mathcal{UL}(s{:}\Psi \mid \mathsf{e}) \overset{\mathsf{U}}{\hookrightarrow} \mathcal{UL}(s'{:}\Psi' \mid \mathsf{e}')}$$

Fig. 7. Multi-language: Lump and Boundaries

The typing rules for language boundaries insert lumps when going from λ^{U} to λ^{L}, and remove them when going back from λ^{L} to λ^{U}. In particular, arbitrary linear types cannot occur at the boundary, they must be of the form $![\sigma]$.

Finally, boundaries have reduction rules: a term or configuration inside a boundary in reduction position is reduced until it becomes a value, and then a lump is added or removed depending on the boundary direction. Note that because the v in $\mathcal{UL}(s : \Psi \mid \mathsf{v})$ is at a duplicable type $![\sigma]$, we know by inversion that the store is empty.

3.2 Interoperability: Static Semantics

If the linear language could not interact with lumped values at all, our multi-language programs would be rather boring, as the only way for the linear extension to provide a value back to ML would be to have received it from λ^{U} and pass it back unchanged (as in the lump embedding of Matthews and Findler [2009]). To provide a real interaction, we provide a way to extract values out of a lump $![\sigma]$, use it at some linear type σ, and put it back in before sending the result to λ^{U}.

$$\textit{Interoperability context } \Sigma ::= \cdot \mid \Sigma, \alpha \simeq !\beta$$

Compatibility relation $\boxed{\Sigma \vdash_{\text{UL}} \sigma \simeq \sigma}$ with $\sigma \simeq \sigma \overset{\text{def}}{=} \cdot \vdash_{\text{UL}} \sigma \simeq \sigma$

$$\frac{}{\Sigma \vdash_{\text{UL}} 1 \simeq !1} \qquad \frac{\Sigma \vdash_{\text{UL}} \sigma_1 \simeq !\sigma_1 \qquad \Sigma \vdash_{\text{UL}} \sigma_2 \simeq !\sigma_2}{\Sigma \vdash_{\text{UL}} \sigma_1 \times \sigma_2 \simeq !(\sigma_1 \otimes \sigma_2)}$$

$$\frac{\Sigma \vdash_{\text{UL}} \sigma_1 \simeq !\sigma_1 \qquad \Sigma \vdash_{\text{UL}} \sigma_2 \simeq !\sigma_2}{\Sigma \vdash_{\text{UL}} \sigma_1 + \sigma_2 \simeq !(\sigma_1 \oplus \sigma_2)} \qquad \frac{\Sigma \vdash_{\text{UL}} \sigma \simeq !\sigma \qquad \Sigma \vdash_{\text{UL}} \sigma' \simeq !\sigma'}{\Sigma \vdash_{\text{UL}} \sigma \to \sigma' \simeq !(!\sigma \multimap !\sigma')}$$

$$\frac{}{\Sigma \vdash_{\text{UL}} \sigma \simeq ![\sigma]} \qquad \frac{\Sigma \vdash_{\text{UL}} \sigma \simeq !\sigma}{\Sigma \vdash_{\text{UL}} \sigma \simeq !!\sigma} \qquad \frac{\Sigma \vdash_{\text{UL}} \sigma \simeq !\sigma}{\Sigma \vdash_{\text{UL}} \sigma \simeq !(\mathsf{Box}\ 1\ \sigma)}$$

$$\frac{\Sigma, \alpha \simeq !\beta \vdash_{\text{UL}} \sigma \simeq !\sigma}{\Sigma \vdash_{\text{UL}} \mu\alpha.\, \sigma \simeq !(\mu\beta.\, \sigma)} \qquad \frac{(\alpha \simeq !\beta) \in \Sigma}{\Sigma \vdash_{\text{UL}} \alpha \simeq !\beta}$$

Interoperability primitives and derived constructs:

$$![\sigma] \underset{\mathsf{lump}^{\sigma}}{\overset{{}^{\sigma}\mathsf{unlump}}{\rightleftarrows}} \sigma \quad \text{whenever} \quad \cdot \vdash_{\text{UL}} \sigma \simeq \sigma \qquad {}^{\sigma}\mathcal{LU}(\mathsf{e}) \overset{\text{def}}{=} {}^{\sigma}\mathsf{unlump}\ \mathcal{LU}(\mathsf{e})$$

$$\mathcal{UL}^{\sigma}(\mathsf{e}) \overset{\text{def}}{=} \mathcal{UL}(\mathsf{lump}^{\sigma}\ \mathsf{e})$$

Fig. 8. Multi-language: Static Interoperability Semantics

The correspondence between intuitionistic types σ and linear types σ is specified by a heterogeneous *compatibility relation* $\sigma \simeq \sigma$ defined in Figure 8 (Multi-language: Static Interoperability Semantics). The specification of this relation is that if $\sigma \simeq \sigma$ holds, then the space of values of $![\sigma]$ and σ are isomorphic: we can convert back and forth between them. When this relation holds, the term-formers lump^{σ} and ${}^{\sigma}\mathsf{unlump}$ perform the conversion. (The position of the index σ emphasizes that the *input* e of $\mathsf{lump}^{\sigma}\ \mathsf{e}$ has type σ, while the *output* of ${}^{\sigma}\mathsf{unlump}\ \mathsf{e}$ has type σ.)

For example, we have $![(\sigma \to \sigma')] \simeq !(![\sigma] \multimap ![\sigma'])$. Given a lumped ML function, we can unlump it to see it as a linear function. We can call it from the linear side, but have to pass it a duplicable argument since an ML function may duplicate its argument. Conversely, we can convert a linear function into a lumped function type to pass it to the ML side, but it has to have a duplicable return type since the ML side may freely share the return value.

Our lump^{σ} and ${}^{\sigma}\mathsf{unlump}$ primitives are only indexed by the linear type σ, because a compatible ML type σ can be uniquely recovered, as per the following result.

Lemma 7 (Determinism of the compatibility relation) *If $\sigma \simeq \sigma$ and $\sigma' \simeq \sigma$ then $\sigma = \sigma'$.*

Proof. By induction on the syntax of σ: the judgment $\Sigma \vdash_{\mathrm{UL}} \sigma \simeq \sigma$ is syntax-directed in its σ component.

Note that the converse property does not hold: for a given σ, there are many σ such that $\sigma \simeq \sigma$. For example, we have $1 \simeq !1$ but also $1 \simeq !!1$. This corresponds to the fact that the linear types are more fine-grained, and make distinctions (inner duplicability, dereference of full locations) that are erased in the ML world. The $\sigma \simeq ![\sigma]$ case also allows you to (un)lump as deeply or as shallowly as you need: $\sigma_1 \times (\sigma_2 \times \sigma_3)$ is compatible with both $!(![\sigma_1] \otimes ![\sigma_2 \times \sigma_3])$ and $!(![\sigma_1] \otimes (![\sigma_2] \otimes ![\sigma_3]))$. We could not systematically translate the complete type σ, as type variables cannot be translated and need to remain lumped. Allowing lumps to "stop" the translation at arbitrary depth is a natural generalization.

Lemma 8 (Substitution of recursive hypotheses) *If $\Sigma, \alpha \simeq !\beta \vdash_{\mathrm{UL}} \sigma \simeq \sigma$, $\alpha \notin \sigma$, and $\Sigma \vdash_{\mathrm{UL}} \sigma' \simeq !\sigma'$ then $\Sigma \vdash_{\mathrm{UL}} \sigma[\sigma'/\alpha] \simeq \sigma[\sigma'/\beta]$.*

Proof. By induction on the $\sigma \simeq \sigma$ derivation. There are two leaf cases: the case recursive hypotheses, which is immediate, and the case of lump $\sigma \simeq ![\sigma]$. In this latter case, notice that σ is a type of λ^{U}, so in particular it does not contain the variable β; and we assumed $\alpha \notin \sigma$ so we also have $\alpha \notin \sigma$, so $\sigma[\sigma'/\alpha] = \sigma \simeq ![\sigma] = \sigma[\sigma'/\beta]$.

The term $\mathcal{LU}(\mathsf{e})$ turns a $\mathsf{e} : \sigma$ into a lumped type $![\sigma]$, and we need to unlump it with some ${}^{\sigma}\mathsf{unlump}$ for a compatible $\sigma \simeq \sigma$ to interact with it on the linear side. It is common to combine both operations and we provide syntactic sugar for it: ${}^{\sigma}\mathcal{LU}(\mathsf{e})$. Similarly $\mathcal{UL}^{\sigma}(\mathsf{e})$ first lumps a linear term then sends the result to the ML world.

3.3 Interoperability: Dynamic Semantics

$$\langle\rangle \leftrightarrow^{!1} \mathsf{share}\,\langle\rangle \qquad \frac{\mathsf{v}_1 \leftrightarrow^{!\sigma_1} \mathsf{share}(s_1 : \Psi_1).\, \mathsf{v}_1 \quad \mathrm{locs}(s_1) = \mathrm{locs}(\Psi_1) = \mathrm{locs}(\mathsf{v}_1) \\ \mathsf{v}_2 \leftrightarrow^{!\sigma_2} \mathsf{share}(s_2 : \Psi_2).\, \mathsf{v}_2 \quad \mathrm{locs}(s_2) = \mathrm{locs}(\Psi_2) = \mathrm{locs}(\mathsf{v}_2)}{\langle \mathsf{v}_1, \mathsf{v}_2\rangle \leftrightarrow^{!(\sigma_1 \otimes \sigma_2)} \mathsf{share}(s_1 \mathbin{+\!\!+} s_2 : \Psi_1 \uplus \Psi_2).\, \langle \mathsf{v}_1, \mathsf{v}_2\rangle}$$

$$\frac{\mathsf{v} \leftrightarrow^{!\sigma_i} \mathsf{share}(s : \Psi).\, \mathsf{v}}{\mathsf{inj_i}\, \mathsf{v} \leftrightarrow^{!(\sigma_1 \oplus \sigma_2)} \mathsf{share}(s : \Psi).\, \mathsf{inj_i}\, \mathsf{v}} \qquad \frac{\sigma \simeq \sigma \qquad \sigma' \simeq \sigma'}{\mathsf{e} \rightarrow^{!(!\sigma \multimap !\sigma')} \mathsf{share}\, \lambda(\mathsf{x} : !\sigma).\, {}^{\sigma'}\mathcal{LU}(\mathsf{e}\, \mathcal{UL}^{\sigma}(\mathsf{x}))}$$

$$\frac{\sigma \simeq \sigma \qquad \sigma' \simeq \sigma'}{\lambda(\mathsf{x} : \sigma).\mathcal{UL}^{\sigma'}(\mathsf{copy}\ \mathsf{e}\ {}^{\sigma}\mathcal{LU}(\mathsf{x})) \leftarrow^{!(\sigma \multimap \sigma')} \mathsf{e}} \qquad \frac{}{\mathsf{v} \leftrightarrow^{![\sigma]} \mathsf{share}\,[\mathsf{v}]}$$

$$\frac{\mathsf{v} \leftrightarrow^{!\sigma} \mathsf{share}\, \mathsf{v}}{\mathsf{v} \leftrightarrow^{!!\sigma} \mathsf{share}\,(\mathsf{share}\, \mathsf{v})} \qquad \frac{\mathsf{v} \leftrightarrow^{!\sigma} \mathsf{share}(s : \Psi).\, \mathsf{v}}{\mathsf{v} \leftrightarrow^{!\mathrm{Box}\ 1\ \sigma} \mathsf{share}([\ell \mapsto (s \mid \mathsf{v})] : (\cdot; \ell \vdash \Psi : \mathrm{Box}\ 1\ \sigma)).\, \ell}$$

$$\frac{\mathsf{v} \leftrightarrow^{!\sigma[\mu\alpha.\sigma/\alpha]} \mathsf{share}(s : \Psi).\, \mathsf{v}}{\mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{v} \leftrightarrow^{!\mu\alpha.\sigma} \mathsf{share}(s : \Psi).\, (\mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{v})}$$

$$(\emptyset \mid \mathsf{share}\,[\mathsf{v}]) \ \overset{{}^{\sigma}\mathsf{unlump}}{\underset{\mathsf{lump}^{\sigma}}{\overset{\mathsf{L}}{\rightsquigarrow} \atop \overset{\mathsf{L}}{\leftsquigarrow}}} \ (\emptyset \mid \mathsf{v}) \qquad \begin{array}{l} \text{whenever } \mathsf{v} \rightarrow^{\sigma} \mathsf{v} \\ \text{whenever } \mathsf{v} \leftarrow^{\sigma} \mathsf{v} \end{array}$$

Fig. 9. Multi-language: Dynamic Interoperability Semantics

We were careful to define the compatibility relation such that $\sigma \simeq \sigma$ only holds when $![\sigma]$ and σ are isomorphic, in the sense that any value of one can be converted into a value of another. Figure 9 defines the operational semantics of the lumping and unlumping operations precisely as realizing these isomorphisms. For concision, we specify the isomorphisms as relations, following the inductive structure of the compatibility judgment itself. We write $(\leftrightarrow)$ when a rule can be read bidirectionally to convert in either directions (assuming the same direction holds of the premises), and $(\leftarrow)$ or $(\rightarrow)$ for rules that only describe how to convert values in one direction.

Lemma 9 (Substitution of polymorphic variables) *If $\Sigma \vdash_{\mathrm{UL}} \sigma \simeq \sigma$ and $\alpha \notin \Sigma$, then $\Sigma \vdash_{\mathrm{UL}} \sigma[\sigma'/\alpha] \simeq \sigma[\sigma'/\alpha]$ and their lumping operations coincide on all values.*

Proof. By induction on $\sigma \simeq \sigma$. In the variable leaf case, we know $\alpha \notin \Sigma$. In the lump leaf case $\sigma \simeq ![\sigma]$, the goal $\sigma[\sigma'/\alpha] \simeq ![\sigma[\sigma'/\alpha]]$ is immediate.

Theorem 3 (Value translations are functional) *If $\sigma \simeq \sigma$, then for any closed value $\mathsf{v} : \sigma$ there is a unique $\mathsf{v} : \sigma$ such that $\mathsf{v} \rightarrow^{\sigma} \mathsf{v}$, and conversely for any closed value $\mathsf{v} : \sigma$ there is a unique $\mathsf{v} : \sigma$ such that $\mathsf{v} \leftarrow^{\sigma} \mathsf{v}$.*

Proof. Remark that in the statement of the term, when we quantify over all closed values $\mathsf{v} : \sigma$, we implicitly assume that in the general case values of v live in the empty global store – otherwise we would have a value of the form $\Psi; \cdot \vdash_{\mathrm{L}} s \mid \mathsf{v} : \sigma$. This is valid because all types σ in the image of the type-compatibility relation are duplicable types of the form $!\sigma'$, so by Lemma 3 (Inversion principle for λ^{L} values) we know that v is in fact of the form $\mathsf{share}(s : \Psi).\, \mathsf{e}'$, living in the empty store.)

The two sides of the result are proved simultaneously by induction on $\sigma \simeq \sigma$, using inversion to reason on the shapes of v and v. Note that the inductive cases remain on closed values: the only variable-binder constructions, λ-abstractions, do not use the recursion hypothesis.

In the recursive case $\mu\alpha.\, \sigma \simeq !(\mu\alpha.\, \sigma)$, to use the induction hypothesis on the folded values we need to know that the unfolded types $\sigma[\mu\alpha.\, \sigma/\alpha]$ and $\sigma[\mu\alpha.\, \sigma/\alpha]$ are compatible. This is exactly Lemma 8 (Substitution of recursive hypotheses), using the hypothesis $\mu\alpha.\, \sigma \simeq !(\mu\alpha.\, \sigma)$ itself.

Lemma 10 (Lumping cancellation) *The lump conversions* lump^{σ} *and* ${}^{\sigma}\mathsf{unlump}$ *cancel each other modulo $\beta\eta$. In particular,*

$$ {}^{\sigma}\mathcal{LU}(\mathcal{UL}^{\sigma}(\mathsf{v})) =_{\beta\eta} \mathsf{v} \qquad\qquad \mathcal{UL}^{\sigma}({}^{\sigma}\mathcal{LU}(\mathsf{v})) =_{\beta\eta} \mathsf{v} $$

Proof. By induction on σ, and then by parallel induction on the derivations of type compatibility and value compatibility. The parallel cases are symmetric by definition, only the function case $!(!\sigma \multimap !\sigma')$ needs to be checked. A simple computation, using the induction hypothesis on the smaller types $!\sigma$ and $!\sigma'$, shows that composing the two function translations gives an η-expansion – plus the $\beta\eta$-steps from the induction hypotheses.

Note that the β-η rules used (for functions and $!\_$) have been proved admissible for the logical relation, so in particular sound with respect with contextual equivalence.

Implementation consideration In a realistic implementation of this multi-language system, we would expect the representation choices made for λ^{U} and λ^{L} to be such that, for some but not all compatible pairs $\sigma \simeq \sigma$, the types σ and σ actually have the exact same representation, making the conversion an efficient no-op. An implementation could even restrict the compatibility relation to accept only the pairs that can be implemented in this way. That is, it would reject some λ^{UL} programs, but the "graceful interoperability" result that is our essential contribution would still hold.

3.4 Full Abstraction from λ^{U} into λ^{UL}

We can now state and prove the major meta-theoretical result of this work, which is the proposed multi-language design extends the simple language λ^{U} in a way that provably has, in a certain sense, "no abstraction leaks".

Definition 1 (Contextual equivalence in λ^{U}) *We say that* e, e' *such that* $\Gamma \vdash_{\mathrm{U}} \mathsf{e}, \mathsf{e}' : \sigma$ *are* contextually equivalent*, written* $\mathsf{e} \approx^{ctx}_{\mathrm{U}} \mathsf{e}'$*, if, for any expression context* $C[\square]$ *such that* $\cdot \vdash_{\mathrm{U}} C[\mathsf{e}] : 1$*, the closed terms* $C[\mathsf{e}]$ *and* $C[\mathsf{e}']$ *are equi-terminating.*

Definition 2 (Contextual equivalence in λ^{UL}) *We say that* e, e' *such that* $\Gamma \vdash_{\mathrm{LU}} \mathsf{e}, \mathsf{e}' : \sigma$ *are* contextually equivalent*, written* $\mathsf{e} \approx^{ctx}_{\mathrm{LU}} \mathsf{e}'$*, if, for any expression context* $C[\square]$ *such that* $\cdot \vdash_{\mathrm{LU}} C[\mathsf{e}] : 1$*, the closed terms* $C[\mathsf{e}]$ *and* $C[\mathsf{e}']$ *are equi-terminating.*

4 Multi-language parametricity

We discussed the design choice of manipulating lumps $[\sigma]$ of any ML type, not just the type variable that motivates them. In the presence of polymorphism, this generalization is also an important design choice to preserve parametricity.

Let us define $\mathsf{id}^{\sigma}(\mathsf{e}) \stackrel{\text{def}}{=} \mathsf{lump}^{\sigma}\ ({}^{\sigma}\mathsf{unlump}\ \mathsf{e})$, and consider a polymorphic term of the form $\Lambda\alpha.\,\mathcal{UL}(\dots \mathsf{id}^{![\alpha]} \dots)$. The (un)lumping operations on a lumped type such as $![\alpha]$ are just the identity: the lumped value is passed around unchanged, so $\mathsf{id}^{![\alpha]}(\mathsf{v})$ will reduce to v. Now, if we instantiate this polymorphic term with a ML type σ, it will reduce to a term $\mathcal{UL}(\dots \mathsf{id}^{![\sigma]} \dots)$ whose unlumping operation is still on a lumped type, so is still exactly the identity.

On the contrary, if we allowed lumps only on type variables, we would have to push the lump inside σ, and the (un)lumping operations would become more complex: if σ starts with an ML product type $\_ \times \_$, it would be turned into a shared linear pair $!(\_ \otimes \_)$ by unlumping, and back into an ML pair by lumping. In general, id^{σ} may perform deep η-expansions of lumped values. The fact that, after instantiation of the polymorphic term, we get a monomorphic term that has different (but η-equivalent) computational behavior would cause meta-theoretic difficulties; this is the approach that was adopted in previous work on multi-languages with polymorphism by Perconti and Ahmed [2014], and it made some of their proofs using a logical relations argument substantially more complicated. In the logical relation, polymorphism is obtained by allowing each polymorphic variable to be replaced by two types related by an "admissible" relation R, and the notion of admissibility of this previous work had to force relations to be compatible with η-expansion, which complicates the proofs.

In contrast, our handling of lump types as turning arbitrary types into black-boxes makes type instantiation obviously parametric. To formally demonstrate this aspect of our design, we develop a step-indexed logical relation (Figure 10) that proves that our multi-language satisfies a strong parametricity property that is not disrupted by the linear sublanguage or the cross-language boundaries.

The logical relation is a family of relations indexed by closed "relational types" which extend the grammar of σ, σ, which include a case for admissible relations R that is used to enable parametric arguments. The step index j in the definitions decreases strictly whenever related values of a type σ or σ are defined in terms of a non-strictly-smaller type; this happens in the definition of the relation at recursive types $\mathcal{V}[\![\mu\alpha.\,\sigma]\!]^j$. Because the language is non-terminating, our relation does not define an equivalence but an approximation: two expressions $(\mathsf{e}_1, \mathsf{e}_1)$ are related in $\mathcal{E}[\![\sigma]\!]^j$ if e_1 approximates e_2: if e_1 reduces to a value in less than j steps, then e_2 must reduce to a related value.

The definition of admissible relations $\mathrm{Rel}[\sigma_1, \sigma_2]$, used to define when λ^{U} values of polymorphic types are related, $\mathcal{V}[\![\forall\alpha.\,\sigma]\!]$, is completely standard, which demonstrates that our notion of boundaries preserves simple parametricity reasoning.

Although we have a stateful linear language, the logical relations for the linear types have more in common with a language with explicit closures—this is another consequence of the remark in Section 2.2 (Linear Memory in λ^{L}) that the language can also be interpreted using a functional semantics. The relations for closed λ^{L} values and expressions, $\mathcal{V}[\![\sigma]\!]$ and $\mathcal{E}[\![\sigma]\!]$, are indexed by a type but do not depend on a store typing: the related values may have different, non-empty store typings. This allows to relate two programs that are equivalent but allocate different references in different ways. Furthermore, since all state is *linear*, we don't need additional machinery to relate stores, since all the values in a store owned by a value will be reflected in the value. For example, the relation for empty locations $\mathcal{V}[\![\mathsf{Box}\ 0]\!]$ relates any two arbitrary (empty) locations, and the relation for non-empty locations $\mathcal{V}[\![\mathsf{Box}\ 1\ \sigma]\!]$ relates possibly-distinct locations that contain related values.

Logical relations effectively translate global invariants of the system into properties of type connectives. For example, consider the reduction rule for lumped values: $\mathcal{UL}(\emptyset : \cdot \mid \mathsf{share}\,[\mathsf{v}]) \overset{\mathsf{U}}{\hookrightarrow} \mathsf{v}$. In this rule we implicitly assumed that a linear value of the shape $[\mathsf{v}]$ at type $![\sigma]$ would occur in an empty local store. The term $\mathsf{share}\,[\mathsf{v}]$ desugars into $\mathsf{share}(\emptyset : \cdot).\,[\mathsf{v}]$, but it is not immediately obvious that this should always be the case since it is possible to compute a value of type $![\sigma]$ by allocating references and using them. The intuitive reason why the store becomes empty when a value $[\mathsf{v}]$ is reached is that linear sub-terms e within v may only occur within a language boundary $\mathcal{UL}(s : \Psi \mid \mathsf{e})$: linear sub-terms have their own local store, so there are no globally visible linear locations for $[\mathsf{v}]$ to refer to. This global reasoning is elegantly expressed in a type-directed way in our logical relation by the definition of related values at lump type, which encodes the invariant that they always have an empty store:

$$\mathcal{V}[\![[\sigma]]\!]^j \overset{\text{def}}{=} \{((\emptyset \mid [\mathsf{v}_1]), (\emptyset \mid [\mathsf{v}_2])) \mid (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\sigma]\!]^j\}$$

From the logical relation defined on closed λ^{UL} terms and values, we define logical approximation relations on open terms $\Gamma \vdash \mathsf{e}_1 \lesssim^{log} \mathsf{e}_2 : \sigma$ and $\Gamma \vdash_{\mathrm{L}} (s_1 \mid \mathsf{e}_1) \lesssim^{log} (s_2 \mid \mathsf{e}_2) : \sigma$ by asking for the open terms and values to be related under all related environments in the standard way—see Appendix A for full details. This lets us demonstrate the Fundamental Property to validate the construction of our logical relation, showing that all typing rules are admissible.

Theorem 4 (Fundamental Property)

1. *If* $!\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma$ *then* $!\Gamma \vdash \mathsf{e} \lesssim^{log} \mathsf{e} : \sigma$

$$
\begin{aligned}
\mathrm{Atom}[\sigma] &\stackrel{\mathrm{def}}{=} \{\mathsf{v} \mid \cdot \vdash_{\mathsf{U}} \mathsf{v} : \sigma\} \\
\mathrm{Rel}[\sigma_1, \sigma_2] &\stackrel{\mathrm{def}}{=} \{\mathsf{R} : \mathbb{N} \to \mathcal{P}(\mathrm{Atom}[\sigma_1] \times \mathrm{Atom}[\sigma_2]) \mid \forall j \leq j'.\ \mathsf{R}^{j'} \subset \mathsf{R}^{j}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}\,[\![\mathsf{R}]\!]^j &\stackrel{\mathrm{def}}{=} \mathsf{R}^j \\
\mathcal{V}\,[\![\sigma_1 \to \sigma_2]\!]^j &\stackrel{\mathrm{def}}{=} \{(\lambda(\mathsf{x}_1 : (\sigma_1)_1).\, \mathsf{e}_1, \lambda(\mathsf{x}_2 : (\sigma_1)_2).\, \mathsf{e}_2) \mid \quad \forall j' \leq j, (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}\,[\![\sigma_1]\!]^{j'}.\ (\mathsf{e}_1[\mathsf{v}_1/\mathsf{x}_1], \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}_2]) \in \mathcal{E}\,[\![\sigma_2]\!]^{j'}\} \\
\mathcal{V}\,[\![\forall\alpha.\, \sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{(\Lambda\alpha.\, \mathsf{v}_1, \Lambda\alpha.\, \mathsf{v}_2) \mid \forall \sigma_1, \sigma_2, \mathsf{R} \in \mathrm{Rel}[\sigma_1, \sigma_2].\ (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}\,[\![\sigma[\mathsf{R}/\alpha]]\!]^j\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}\,[\![\mathsf{1}]\!]^j &\stackrel{\mathrm{def}}{=} \{((\emptyset \mid \langle\rangle), (\emptyset \mid \langle\rangle))\} \\
\mathcal{V}\,[\![\sigma \otimes \sigma']\!]^j &\stackrel{\mathrm{def}}{=} \{((s_1 +\!\!\!+ s_1' \mid \langle \mathsf{v}_1, \mathsf{v}_1' \rangle), (s_2 +\!\!\!+ s_2' \mid \langle \mathsf{v}_2, \mathsf{v}_2' \rangle)) \mid \quad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma]\!]^j \wedge ((s_1' \mid \mathsf{v}_1'), (s_2' \mid \mathsf{v}_2')) \in \mathcal{V}\,[\![\sigma']\!]^j\} \\
\mathcal{V}\,[\![\sigma' \multimap \sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{((s_1 \mid \lambda(\mathsf{x} : \sigma').\, \mathsf{e}_1), (s_2 \mid \lambda(\mathsf{x} : \sigma').\, \mathsf{e}_2)) \mid \\
&\qquad \forall j' \leq j, s_1', s_2', ((s_1'' \mid \mathsf{v}_1), (s_2'' \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma']\!]^{j'}. \\
&\qquad\quad s_1' = s_1 +\!\!\!+ s_1'' \wedge s_2' = s_2 +\!\!\!+ s_2'' \Rightarrow ((s_1' \mid \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}]), (s_2' \mid \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}])) \in \mathcal{E}\,[\![\sigma]\!]^{j'}\} \\
\mathcal{V}\,[\![\mu\alpha.\, \sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{((s_1 \mid \mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{v}_1), (s_2 \mid \mathsf{fold}_{\mu\alpha.\sigma}\, \mathsf{v}_2)) \mid \quad \forall j' < j. ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma[\mu\alpha.\, \sigma/\alpha]]\!]^{j'}\} \\
\mathcal{V}\,[\![!\sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{((\emptyset \mid \mathsf{share}(s_1 : \Psi_1).\, \mathsf{v}_1), (\emptyset \mid \mathsf{share}(s_2 : \Psi_2).\, \mathsf{v}_2)) \mid \quad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma]\!]^j\} \\
\mathcal{V}\,[\![\mathsf{Box}\ 0]\!]^j &\stackrel{\mathrm{def}}{=} \{(([\ell_1 \mapsto \cdot] \mid \ell_1), ([\ell_2 \mapsto \cdot] \mid \ell_2))\} \\
\mathcal{V}\,[\![\mathsf{Box}\ 1\ \sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{(([\ell_1 \mapsto (s_1 \mid \mathsf{v}_1)] \mid \ell_1), ([\ell_2 \mapsto (s_2 \mid \mathsf{v}_2)] \mid \ell_2)) \mid \quad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma]\!]^j\} \\
\mathcal{V}\,[\![[\sigma]]\!]^j &\stackrel{\mathrm{def}}{=} \{((\emptyset \mid [\mathsf{v}_1]), (\emptyset \mid [\mathsf{v}_2])) \mid (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}\,[\![\sigma]\!]^j\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}\,[\![\sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{(\mathsf{e}_1, \mathsf{e}_2) \mid \forall j' \leq j.\ \mathsf{e}_1 \overset{\mathsf{U}}{\hookrightarrow}^{j'} \mathsf{v}_1 \Rightarrow \exists \mathsf{v}_2.\ \mathsf{e}_2 \overset{\mathsf{U}}{\hookrightarrow}^{*} \mathsf{v}_2 \wedge (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}\,[\![\sigma]\!]^{j-j'}\} \\
\mathcal{E}\,[\![\sigma]\!]^j &\stackrel{\mathrm{def}}{=} \{((s_1 \mid \mathsf{e}_1), (s_2 \mid \mathsf{e}_2)) \mid \quad \forall j' \leq j, (s_1' \mid \mathsf{v}_1).(s_1 \mid \mathsf{e}_1) \overset{\mathsf{L}}{\hookrightarrow}^{j'} (s_1' \mid \mathsf{v}_1) \Rightarrow \\
&\qquad \exists (s_2' \mid \mathsf{v}_2).(s_2 \mid \mathsf{e}_2) \overset{\mathsf{L}}{\hookrightarrow}^{*} (s_2' \mid \mathsf{v}_2) \ \wedge\ ((s_1' \mid \mathsf{v}_1), (s_2' \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![\sigma]\!]^{j-j'}\}
\end{aligned}
$$

Fig. 10. Multi-language Logical Relation (excerpt)

2. If $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ *then* $\Gamma \vdash_{\mathrm{L}} (s \mid \mathsf{e}) \lesssim^{log} (s \mid \mathsf{e}) : \sigma$

We also prove that the logical relation is sound with respect to contextual equivalence. For this, we define contextual approximation relations $\Gamma \vdash \mathsf{e}_1 \lesssim^{ctx} \mathsf{e}_2 : \sigma$ and $\Gamma \vdash_{\mathrm{L}} (s_1 \mid \mathsf{e}_1) \lesssim^{ctx} (s_2 \mid \mathsf{e}_2) : \sigma$, by asking that $\mathsf{e}_1, (s_1 \mid \mathsf{e}_1)$ terminate more often than $\mathsf{e}_2, (s_2 \mid \mathsf{e}_2)$ when run under arbitrary contexts—see Appendix A for details.

Theorem 5 (Soundness of Logical Relation with respect to Contextual Equivalence) $(\lesssim^{log}) \subset (\lesssim^{ctx})$.

5 Hybrid program examples

5.1 In-Place Transformations

In Section 2.2 (Linear Memory in λ^{L}) we proposed a program for in-place reversal of linear lists defined by the type $\mathsf{LinList}\ \sigma \stackrel{\text{def}}{=} \mu\alpha.\, 1 \oplus \mathsf{Box}\ 1\ (\sigma \otimes \alpha)$. We can also define a type of ML lists $\mathsf{List}\ \sigma \stackrel{\text{def}}{=} \mu\alpha.\, 1 + \sigma \times \alpha$. Note that ML lists are compatible with shared linear lists, in the sense that $\mathsf{List}\ \sigma \simeq\ !(\mathsf{LinList}\ ![\sigma])$. This enables writing in-place list-manipulation functions in λ^{L}, and exposing them to beginners at a λ^{U} type:

$$\mathsf{rev}\ \mathsf{xs} \stackrel{\text{def}}{=} \mathcal{UL}^{\mathsf{LinList}\ ![\sigma]}(\mathsf{share}\,(\mathsf{rev\_into}\ \mathsf{copy}\ ({}^{\mathsf{LinList}\ ![\sigma]}\mathcal{LU}(\mathsf{xs}))\ \mathsf{Nil}))$$

This example is arguably silly, as the allocations that are avoided by doing an in-place traversal are paid when copying the shared list to obtain a uniquely-owned version. A better example of list operations that can profitably be sent on the linear side is quicksort, whose code we give in Figure 11 (Quicksort). An ML implementation allocates intermediary lists for each recursive call, while the surprisingly readable λ^{U} implementation only allocates for the first copy.

5.2 Typestate Protocols

Linear types can enforce proper allocation and deallocation of resources, and in general any automata/typestate-like protocols on their usage by encoding the state transitions as linear transformations. In the simple example of file-descriptor handling in the introduction, additional safety compared to ML programming can be obtained by exposing file-handling functions on the λ^{U} side, with linear types. We assumed the following API for linear file handling, which enforces a correct usage protocol:

$$\begin{aligned}
\mathsf{open} &: !(![\mathsf{Path}] \multimap \mathsf{Handle})\\
\mathsf{line} &: !(\mathsf{Handle} \multimap (\mathsf{Handle} \oplus (![\mathsf{String}] \otimes \mathsf{Handle})))\\
\mathsf{close} &: !(\mathsf{Handle} \multimap 1)
\end{aligned}$$

Another interesting example of protocol usage for which linear types help is the use of *transient* versions of persistent data structures, as popularized by Clojure. An unrestricted type $\mathsf{Set}\ \alpha$ may represent persistent sets as balanced trees

```
partition : !(!α ⊸ Bool) ⊸ LList !α ⊸ LList !α ⊕ LList !α
partition p li = partition_aux p (Nil, Nil) li
partition_aux p (yes, no) Nil = (yes, no)
partition_aux p (yes, no) (Cons l x xs) =
  let (yes, no) =
    if copy p x
    then (Cons l x yes, no)
    else (yes, Cons l x no) in
  partition_aux p (yes, no) xs

lin_quicksort : LList !α ⊸ LList !α
lin_quicksort li = quicksort_aux li Nil
quicksort_aux Nil acc = acc
quicksort_aux (Cons l head li) acc =
  let p = share (fun x -> x < head) in
  let (below, above) = partition p li in
  quicksort_aux below (Cons l head (quicksort_aux above acc))

quicksort li UL(li) = UL(share (lin_quicksort (copy li)))
```

Fig. 11. Quicksort

with logarithmic operations performing path-copying. A transient call returns a mutable version of the structure that supports efficient batch in-place updates, before a persistent call freezes this transient structure back into a persistent tree. To preserve a purely functional semantics, we must enforce that the intermediate transient value is uniquely owned. We can do this by using the linear types for the transient API:

```
type Set α
val add : Set α → α → Set α (* path copy *)
...
type MutSet α
val add: !(MutSet α ⊸ α ⊸ MutSet α) (* in-place update *)
...
val transient : !(![Set α] ⊸ MutSet ![α])
val persistent : !(MutSet ![α] ⊸ ![Set α])
```

6 Conclusion

In our proposed multi-language design, a simple linear type system mirroring the standard rules of intuitionistic linear logic can be equipped with linear state and usefully complement a general-purpose functional ML language, without breaking equational reasoning or parametricity—and without requiring a significantly more complex meta-theory.

Fine-grained language boundaries allow interesting programming patterns to emerge, and full abstraction provides a novel rigorous specification of what it means for multi-language design to avoid *abstraction leaks* from advanced features into the general-purpose or beginner-friendly languages.

6.1 Related Work

Having a stack of usable, interoperable languages, extensions or dialects is at the forefront of the Racket approach to programming environments, in particular for teaching [Felleisen, Findler, Flatt, and Krishnamurthi, 2004].

Our multi-language semantics builds on the seminal work by Matthews and Findler [2009], who gave a formal semantics of interoperability between a dynamically and a statically typed language. Others have followed the Matthews-Findler approach of designing multi-language systems with fine-grained boundaries—for instance, formalizing interoperability between a simply and dependently typed language [Osera, Sjöberg, and Zdancewic, 2012]; between a functional and typed assembly language [Patterson, Perconti, Dimoulas, and Ahmed, 2017]; between an ML-like and an affinely typed language, where linearity is enforced at runtime on the ML side using stateful contracts [Tov and Pucella, 2010]; and between the source and target languages of compilation to specify compiler correctness [Perconti and Ahmed, 2014]. However, all these papers address only the question of soundness of the multi-language; we propose a formal treatment of *usability* and absence of abstraction leaks.

The only work to establish that a language embeds into a multi-language in a fully abstract way is the work on fully abstract compilation by Ahmed and Blume [2011] and New, Bowman, and Ahmed [2016] who show that their compiler's source language embeds into their source-target multi-language in a fully abstract way. But the focus of this work was on fully abstract compilation, not on usability of user-facing languages.

The Eco project [Barrett, Bolz, Diekmann, and Tratt, 2016] is studying multi-language systems where user-exposed languages are combined in a very fine-grained way; it is closely related in that it studies the user experience in a multi-language system. The choice of an existing dynamic language creates delicate interoperability issues (conflicting variable scoping rules, etc.) as well as performance challenges. We propose a different approach, to design new multi-languages from scratch with interoperability in mind to avoid legacy obstacles.

We are not aware of existing systems exploiting the simple idea of using promotion to capture uniquely-owned state and dereliction to copy it—common formulations would rather perform copies on the contraction rule.

The general idea that linear types can permit reuse of unused allocated cells is not new. In Wadler [1990], a system is proposed with both linear and non-linear types to attack precisely this problem. It is however more distant from standard linear logic and somewhat ad-hoc; for example, there is no way to permanently turn a uniquely-owned value into a shared value, it provides instead a local *borrowing* construction that comes with ad-hoc restrictions necessary for safety. (The inability to *give up* unique ownership, which is essential in our list-programming examples, seems to also be missing from Rust, where one would need to perform a costly operation of traversing the graph of the value to turn all pointers into `Arc` nodes.)

The RAML project [Hoffmann, Aehlig, and Hofmann, 2012] also combines linear logic and memory reuse: its *destructive match* operator will implicitly reuse consumed cells in new allocations occurring within the match body. Multi-languages give us the option to explore more explicit, flexible representations of those low-level concern, without imposing the complexity to all programmers.

A recent related work is the Cogent language [O'Connor, Chen, Rizkallah, Amani, Lim, Murray, Nagashima, Sewell, and Klein, 2016], in which linear state is also viewed as both functional and imperative – the latter view enabling memory reuse. The language design is interestingly reversed: in Cogent, the linear layer is the simple language that everyone uses, and the non-linear language is a complex but powerful language that is used when one really has to, named C.

Our linear language λ^{L} is sensibly simpler, and in several ways less expressive, than advanced programming languages based on linear logic [Tov and Pucella, 2011], separation logic [Balabonski, Pottier, and Protzenko, 2016], fine-grained permissions [Garcia, Tanter, Wolff, and Aldrich, 2014]: it is not designed to stand on its own, but to serve as a useful side-kick to a functional language, allowing safer resource handling.

One major simplification of our design compared to more advanced linear or separation-logic-based languages is that we do not separate physical locations from the logical capability/permission to access them (e.g., as in Ahmed, Fluet, and Morrisett [2007]). This restricts expressiveness in well-understood ways [Fahndrich and DeLine, 2002]: shared values cannot point to linear values.

Alms [Tov and Pucella, 2011], Quill [Morris, 2016] and Linear Haskell [Bernardy, Boespflug, Newton, Jones, and Spiwack, 2018] add linear types to a functional language, trying hard not to lose desirable usability property, such as type inference or the genericity of polymorphic higher-order functions. This is very challenging; for example, Linear Haskell gives up on principality of inference[7]. Our multi-language design side-steps this issue as the general-purpose language remains unchanged. Language boundaries are more rigid than an ideal no-compromise language, as they force users to preserve the distinction between the general-purpose and the advanced features; it is precisely this compromise that gives a design of reduced complexity.

Finally, on the side of the semantics, our system is related to LNL [Benton, 1994], a calculus for linear logic that, in a sense, is itself built as a multi-language system where (non-duplicable) linear types and (duplicable) intuitionistic types interact through a boundary. It is not surprising that our design contains an instance of this adjunction: for any σ there is a unique σ such that $\sigma \simeq\, !\sigma$, and converting a σ value to this σ and back gives a $!\sigma$ and is provably equivalent, by boundary cancellation, to just using share.

[7] Thanks to Stephen Dolan for pointing out that $\lambda f.\lambda x.\, f\ x$ has several incompatible Linear Haskell types.

Acknowledgments

We thank our anonymous reviewers for their feedback, as well as Neelakantan Krishnaswami, François Pottier, Jennifer Paykin, Sylvie Boldo and Simon Peyton-Jones for our discussions on this work.

This work was supported in part by the National Science Foundation under grants CCF-1422133 and CCF-1453796, and the European Research Council under ERC Starting Grant SECOMP (715753). Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of our funding agencies.

Bibliography

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. 2

Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 157–168, September 2008. 2

Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, September 2011. 2, 27

Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3 : A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007. 28

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, 38(4):14:1–14:94, August 2016. 28

Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained language composition: A case study. In *ECOOP*, July 2016. 16, 27

P Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic*, 1994. 28

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018. doi: 10.1145/3158093. URL http://doi.acm.org/10.1145/3158093. 28

Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 328–342, 1992. 2

Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, 2016. 2

Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI'02*, PLDI '02, 2002. 28

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004. 27

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *TOPLAS*, 2014. 28

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012. 28

Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *European Symposium on Programming (ESOP)*, 2005. 2

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):12, 2009. 18, 27

A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 191–203, 1988. 2

Robin Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1), 1977. 2

J. Garrett Morris. The best of both worlds: Linear functional programming without compromise. In *ICFP*, 2016. 28

Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*, September 2016. 2, 27

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, 2016. 9, 28

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *Programming Languages meets Program Verification (PLPV)*, January 2012. 27

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6:1–6:50, April 2015. 2

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*, June 2017. To appear. Available at `http://www.ccs.neu.edu/home/amal/papers/funtal.pdf`. 27

James T Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014. 22, 27

Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, 1993. 32

Jesse Tov and Riccardo Pucella. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, March 2010. 27

Jesse A Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011. 28

Philip Wadler. Linear Types Can Change the World! In *Programming Concepts and Methods*, 1990. 27

A Logical relation

To define the logical relation in a precise way, we introduce a new grammar of "relational types" ρ, ρ, that simply extend the grammar of σ, σ to include a case for a relation on λ^{U} types $(\mathsf{R}, \sigma_1, \sigma_2)$.

Introducing this lightweight syntax for relations here is a middle way between definitions that use an explicit relational substitution (cluttering the relation with yet another index) and a full-blown logic for parametricity as in Plotkin and Abadi [1993]

$$\begin{aligned} \rho &::= \ldots \text{all cases for } \sigma\text{, but using } \rho, \rho \text{ recursively} \mid (\mathsf{R}, \sigma_1, \sigma_2) \\ \rho &::= \text{all cases for } \sigma\text{, but using } \rho, \rho \text{ recursively} \end{aligned}$$

Fig. 12. Relation Type Syntax

Every ρ has two associated types, the types of terms that it relates, which we denote $(\rho)_1, (\rho)_2$. It is defined as follows:

$$\begin{aligned} ((\mathsf{R}, \sigma_1, \sigma_2))_1 &\stackrel{\text{def}}{=} \sigma_1 & (\rho_1 \otimes \rho_2)_i &\stackrel{\text{def}}{=} \rho_1 \otimes \rho_2 \\ ((\mathsf{R}, \sigma_1, \sigma_2))_2 &\stackrel{\text{def}}{=} \sigma_2 & (1)_i &\stackrel{\text{def}}{=} 1 \\ (\alpha)_i &\stackrel{\text{def}}{=} \alpha & (\rho_1 \multimap \rho_2)_i &\stackrel{\text{def}}{=} (\rho_1)_i \multimap (\rho_2)_i \\ (\rho_1 \times \rho_2)_i &\stackrel{\text{def}}{=} (\rho_1)_i \times (\rho_2)_i & (\rho_1 \oplus \rho_2)_i &\stackrel{\text{def}}{=} (\rho_1)_i \oplus (\rho_2)_i \\ (1)_i &\stackrel{\text{def}}{=} 1 & (\mu\alpha.\, \rho)_i &\stackrel{\text{def}}{=} \mu\alpha.\, (\rho)_i \\ (\rho_1 \to \rho_2)_i &\stackrel{\text{def}}{=} (\rho_1)_i \to (\rho_2)_i & (\alpha)_i &\stackrel{\text{def}}{=} \alpha \\ (\rho_1 + \rho_2)_i &\stackrel{\text{def}}{=} (\rho_1)_i + (\rho_2)_i & (!\rho)_i &\stackrel{\text{def}}{=} !(\rho)_i \\ (\mu\alpha.\, \rho)_i &\stackrel{\text{def}}{=} \mu\alpha.\, (\rho)_i & (\mathsf{Box}\ 1\ \rho)_i &\stackrel{\text{def}}{=} \mathsf{Box}\ 1\ (\rho)_i \\ (\forall\alpha.\, \rho)_i &\stackrel{\text{def}}{=} \forall\alpha.\, (\rho)_i & (\mathsf{Box}\ 0\ )_i &\stackrel{\text{def}}{=} \mathsf{Box}\ 0 \end{aligned}$$

First, we define when closed values are related at each type, indexing by a natural number to break the circularity of recursive types. The relations are defined by nested induction on j, ρ, ρ, any time a bigger type is used in a definition, the step-index j is decremented.

The definition of $\mathcal{E}\left[\!\left[\sigma\right]\!\right]^j$ shows that this is an assymetric relation capturing a notion of *approximation*, not equivalence.

Next, we extend the relations to open terms by defining open terms to be related when they are related when closed by related substitutions.

The Fundamental Property is the key to proving parametricity results.

Lemma 11 (Fundamental Property)

1. *If* $!\Gamma \vdash_{\mathrm{U}} \mathsf{v} : \sigma$ *then* $!\Gamma \vdash_v \mathsf{v} \lesssim^{log} \mathsf{v} : \sigma$
2. *If* $!\Gamma \vdash_{\mathrm{U}} \mathsf{e} : \sigma$ *then* $!\Gamma \vdash \mathsf{e} \lesssim^{log} \mathsf{e} : \sigma$
3. *If* $\Psi; \Gamma \vdash_{\mathrm{L}} s \mid \mathsf{e} : \sigma$ *then* $!\Gamma \vdash_{\mathrm{L}} (s \mid \mathsf{e}) \lesssim^{log} (s \mid \mathsf{e}) : \sigma$

$$\mathrm{Atom}[\sigma] \stackrel{\text{def}}{=} \{\mathsf{v} \mid \cdot \vdash_{\mathrm{U}} \mathsf{v} : \sigma\}$$
$$\mathrm{Rel}[\sigma_1, \sigma_2] \stackrel{\text{def}}{=} \{\mathsf{R} : \mathbb{N} \to \mathcal{P}(\mathrm{Atom}[\sigma_1] \times \mathrm{Atom}[\sigma_2]) \mid \forall j \leq j'.\ \mathsf{R}^{j'} \subset \mathsf{R}^{j}\}$$

$$\begin{aligned}
\mathcal{V}[\![(\mathsf{R}, \sigma_1, \sigma_2)]\!]^j &\stackrel{\text{def}}{=} \mathsf{R}^j \\
\mathcal{V}[\![1]\!]^j &\stackrel{\text{def}}{=} \{(\langle\rangle, \langle\rangle)\} \\
\mathcal{V}[\![\rho \times \rho']\!]^j &\stackrel{\text{def}}{=} \{(\langle \mathsf{v}_1, \mathsf{v}_1'\rangle, \langle \mathsf{v}_2, \mathsf{v}_2'\rangle) \mid (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho]\!]^j \wedge (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho]\!]^j\} \\
\mathcal{V}[\![\rho_1 + \rho_2]\!]^j &\stackrel{\text{def}}{=} \{(\mathsf{inj_i}\, \mathsf{v}_1, \mathsf{inj_i}\, \mathsf{v}_2) \mid (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho_i]\!]^j\} \\
\mathcal{V}[\![\mu\alpha.\, \rho]\!]^j &\stackrel{\text{def}}{=} \{(\mathsf{fold}_{(\mu\alpha.\rho)_1}\, \mathsf{v}_1, \mathsf{fold}_{(\mu\alpha.\rho)_2}\, \mathsf{v}_1) \mid \forall j' < j.\ (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho[\mu\alpha.\, \rho/\alpha]]\!]^{j'}\} \\
\mathcal{V}[\![\rho_1 \to \rho_2]\!]^j &\stackrel{\text{def}}{=} \{(\lambda(\mathsf{x}_1 : (\rho_1)_1).\, \mathsf{e}_1, \lambda(\mathsf{x}_2 : (\rho_1)_2).\, \mathsf{e}_2) \mid \\
&\qquad \forall j' \leq j, (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho_1]\!]^{j'}.\ (\mathsf{e}_1[\mathsf{v}_1/\mathsf{x}_1], \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}_2]) \in \mathcal{E}[\![\rho_2]\!]^{j'}\} \\
\mathcal{V}[\![\forall\alpha.\, \rho]\!]^j &\stackrel{\text{def}}{=} \{(\Lambda\alpha.\, \mathsf{v}_1, \Lambda\alpha.\, \mathsf{v}_2) \mid \forall \sigma_1, \sigma_2, \mathsf{R} \in \mathrm{Rel}[\sigma_1, \sigma_2].\ (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho[(\mathsf{R}, \sigma_1, \sigma_2)/\alpha]]\!]^j\} \\
\mathcal{V}[\![1]\!]^j &\stackrel{\text{def}}{=} \{((\emptyset \mid \langle\rangle), (\emptyset \mid \langle\rangle))\} \\
\mathcal{V}[\![\rho \otimes \rho']\!]^j &\stackrel{\text{def}}{=} \{((s_1 +\!\!\!+ s_1' \mid \langle \mathsf{v}_1, \mathsf{v}_1'\rangle), (s_2 +\!\!\!+ s_2' \mid \langle \mathsf{v}_2, \mathsf{v}_2'\rangle)) \mid \\
&\qquad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho]\!]^j \wedge ((s_1' \mid \mathsf{v}_1'), (s_2' \mid \mathsf{v}_2')) \in \mathcal{V}[\![\rho']\!]^j\} \\
\mathcal{V}[\![\rho_1 \oplus \rho_2]\!]^j &\stackrel{\text{def}}{=} \{((s_1 \mid \mathsf{inj_i}\, \mathsf{v}_1), (s_2 \mid \mathsf{inj_i}\, \mathsf{v}_2)) \mid \\
&\qquad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho_i]\!]^j\} \\
\mathcal{V}[\![\mu\alpha.\, \rho]\!]^j &\stackrel{\text{def}}{=} \{((s_1 \mid \mathsf{fold}_{\mu\alpha.\rho}\, \mathsf{v}_1), (s_2 \mid \mathsf{fold}_{\mu\alpha.\rho}\, \mathsf{v}_2)) \mid \\
&\qquad \forall j' < j.((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho[\mu\alpha.\, \rho/\alpha]]\!]^{j'}\} \\
\mathcal{V}[\![\rho' \multimap \rho]\!]^j &\stackrel{\text{def}}{=} \{((s_1 \mid \lambda(\mathsf{x} : \rho').\, \mathsf{e}_1), (s_2 \mid \lambda(\mathsf{x} : \rho').\, \mathsf{e}_2)) \mid \\
&\qquad \forall j' \leq j, s_1', s_2', ((s_1'' \mid \mathsf{v}_1), (s_2'' \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho']\!]^{j'}. \\
&\qquad\quad s_1' = s_1 +\!\!\!+ s_1'' \wedge s_2' = s_2 +\!\!\!+ s_2'' \Rightarrow \\
&\qquad\quad ((s_1' \mid \mathsf{e}_1[\mathsf{v}_1/\mathsf{x}]), (s_2' \mid \mathsf{e}_2[\mathsf{v}_2/\mathsf{x}])) \in \mathcal{E}[\![\rho]\!]^{j'}\} \\
\mathcal{V}[\![!\rho]\!]^j &\stackrel{\text{def}}{=} \{((\emptyset \mid \mathsf{share}(s_1 : \Psi_1).\, \mathsf{v}_1), (\emptyset \mid \mathsf{share}(s_2 : \Psi_2).\, \mathsf{v}_2)) \mid \\
&\qquad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho]\!]^j\} \\
\mathcal{V}[\![\mathsf{Box}\ 0]\!]^j &\stackrel{\text{def}}{=} \{(([\ell_1 \mapsto \cdot] \mid \ell_1), ([\ell_2 \mapsto \cdot] \mid \ell_2))\} \\
\mathcal{V}[\![\mathsf{Box}\ 1\ \rho]\!]^j &\stackrel{\text{def}}{=} \{(([\ell_1 \mapsto (s_1 \mid \mathsf{v}_1)] \mid \ell_1), ([\ell_2 \mapsto (s_2 \mid \mathsf{v}_2)] \mid \ell_2)) \mid \\
&\qquad ((s_1 \mid \mathsf{v}_1), (s_2 \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho]\!]^j\} \\
\mathcal{V}[\![[\rho]]\!]^j &\stackrel{\text{def}}{=} \{((\emptyset \mid [\mathsf{v}_1]), (\emptyset \mid [\mathsf{v}_2])) \mid (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho]\!]^j\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![\rho]\!]^j &\stackrel{\text{def}}{=} \{(\mathsf{e}_1, \mathsf{e}_2) \mid \forall j' \leq j.\ \mathsf{e}_1 \overset{\mathsf{U}}{\hookrightarrow}^{j'} \mathsf{v}_1 \Rightarrow \\
&\qquad \exists \mathsf{v}_2.\ \mathsf{e}_2 \overset{\mathsf{U}}{\hookrightarrow}^{*} \mathsf{v}_2 \wedge (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\rho]\!]^{j-j'}\} \\
\mathcal{E}[\![\rho]\!]^j &\stackrel{\text{def}}{=} \{((s_1 \mid \mathsf{e}_1), (s_2 \mid \mathsf{e}_2)) \mid \\
&\qquad \forall j' \leq j, (s_1' \mid \mathsf{v}_1).(s_1 \mid \mathsf{e}_1) \overset{\mathsf{L}}{\hookrightarrow}^{j'} (s_1' \mid \mathsf{v}_1) \Rightarrow \\
&\qquad \exists (s_2' \mid \mathsf{v}_2).(s_2 \mid \mathsf{e}_2) \overset{\mathsf{L}}{\hookrightarrow}^{*} (s_2' \mid \mathsf{v}_2) \wedge \\
&\qquad ((s_1' \mid \mathsf{v}_1), (s_2' \mid \mathsf{v}_2)) \in \mathcal{V}[\![\rho]\!]^{j-j'}\}
\end{aligned}$$

Fig. 13. Logical Approximation for Closed Terms

$$
\begin{aligned}
\mathcal{G}\,[\![\cdot]\!]^j &\stackrel{\text{def}}{=} \{((\emptyset,\emptyset) \mid \varnothing)\} \\
\mathcal{G}\,[\![\Gamma, \mathsf{x}{:}\sigma]\!]^j &\stackrel{\text{def}}{=} \{((s_1 +\!\!+\, s_1', s_2 +\!\!+\, s_2') \mid \gamma[\mathsf{x} \mapsto (\mathsf{v}_1, \mathsf{v}_2)]) \mid \\
&\qquad ((s_1, s_2) \mid \gamma) \in \mathcal{G}\,[\![\Gamma]\!]^j \wedge ((s_1' \mid \mathsf{v}_1), (s_2' \mid \mathsf{v}_2)) \in \mathcal{V}\,[\![(\gamma)_R(\sigma)]\!]^j\} \\
\mathcal{G}\,[\![\Gamma, \mathsf{x}{:}\sigma]\!]^j &\stackrel{\text{def}}{=} \{((s_1, s_2) \mid \gamma[\mathsf{x} \mapsto (\mathsf{v}_1, \mathsf{v}_2)]) \mid \\
&\qquad ((s_1, s_2) \mid \gamma) \in \mathcal{G}\,[\![\Gamma]\!]^j \wedge (\mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}\,[\![(\gamma)_R(\sigma)]\!]^j\} \\
\mathcal{G}\,[\![\Gamma, \alpha]\!]^j &\stackrel{\text{def}}{=} \{((s_1, s_2) \mid \gamma[\alpha \mapsto (\mathsf{R}, \sigma_1, \sigma_2)]) \mid \\
&\qquad \mathsf{R} \in \mathrm{Rel}[\sigma_1, \sigma_2] \wedge ((s_1, s_2) \mid \gamma) \in \mathcal{G}\,[\![\Gamma]\!]^j\}
\end{aligned}
$$

$$
!\Gamma \vdash \mathsf{e}_1 \lesssim^{log} \mathsf{e}_2 : \sigma \stackrel{\text{def}}{=} \forall j \geq 0, ((\emptyset,\emptyset) \mid \gamma) \in \mathcal{G}\,[\![!\Gamma]\!]^j \,.((\gamma)_1(\mathsf{e}_1), (\gamma)_2(\mathsf{e}_2)) \in \mathcal{E}\,[\![(\gamma)_R(\sigma)]\!]^j
$$

$$
!\Gamma \vdash_v \mathsf{v}_1 \lesssim^{log} \mathsf{v}_2 : \sigma \stackrel{\text{def}}{=} \forall j \geq 0, ((\emptyset,\emptyset) \mid \gamma) \in \mathcal{G}\,[\![!\Gamma]\!]^j \,.((\gamma)_1(\mathsf{v}_1), (\gamma)_2(\mathsf{v}_2)) \in \mathcal{V}\,[\![(\gamma)_R(\sigma)]\!]^j
$$

$$
\begin{gathered}
\Gamma \vdash_{\mathrm{L}} (s_1 \mid \mathsf{e}_1) \lesssim^{log} (s_2 \mid \mathsf{e}_2) : \sigma \stackrel{\text{def}}{=} \\
\begin{aligned}
&\forall j \geq 0, ((s_1', s_2') \mid \gamma) \in \mathcal{G}\,[\![\Gamma]\!]^j \,. \\
&\quad ((s_1 +\!\!+\, s_1' \mid (\gamma)_1(\mathsf{e}_1)), (s_2 +\!\!+\, s_2' \mid (\gamma)_2(\mathsf{e}_2))) \in \mathcal{E}\,[\![(\gamma)_R(\sigma)]\!]^j
\end{aligned}
\end{gathered}
$$

Fig. 14. Logical Approximation for Open Terms

$$
!\Gamma \vdash \mathsf{e}_1 \lesssim^{ctx} \mathsf{e}_2 : \sigma \stackrel{\text{def}}{=} \forall C.\ \cdot \vdash_{\mathrm{U}} C[\mathsf{e}_1] : \mathbf{1} \wedge \cdot \vdash_{\mathrm{U}} C[\mathsf{e}_2] : \mathbf{1} \wedge C[\mathsf{e}_1] \stackrel{\mathsf{U}}{\hookrightarrow}^* \langle\rangle \implies C[\mathsf{e}_2] \stackrel{\mathsf{U}}{\hookrightarrow}^* \langle\rangle
$$

$$
\begin{gathered}
\Gamma \vdash_{\mathrm{L}} (s_1 \mid \mathsf{e}_1) \lesssim^{ctx} (s_2 \mid \mathsf{e}_2) : \sigma \stackrel{\text{def}}{=} \forall C.\ \cdot \vdash_{\mathrm{U}} C[(s_1 \mid \mathsf{e}_1)] : \mathbf{1} \wedge \cdot \vdash_{\mathrm{U}} C[\mathsf{e}_1](s_2 \mid \mathsf{e}_2) : \\
\mathbf{1} \wedge C[(s_1 \mid \mathsf{e}_1)] \stackrel{\mathsf{U}}{\hookrightarrow}^* \langle\rangle \implies C[(s_2 \mid \mathsf{e}_2)] \stackrel{\mathsf{U}}{\hookrightarrow}^* \langle\rangle
\end{gathered}
$$

Fig. 15. Contextual Approximation

Finally, we prove our logical relation is sound with respect to contextual equivalence, that is, it can be used as a more tractible way to prove contextual equivalence results, such as lump/unlump cancellation.

Theorem 6 (Soundness of Logical Relation) $\lesssim^{log} \subset \lesssim^{ctx}$

The proof is by induction on contexts, showing that every term formation rule preserves logical relatedness. These "compatibility" lemmas are extensive, but their proofs are simple. Their proofs are in the extended technical report.